

THE UNIVERSITY OF NEWCASTLE UPON TYNE
DEPARTMENT OF COMPUTING SCIENCE

Dataflow Development of Medium-Grained Parallel Software

by
Jonathan William Harley

PhD Thesis

September 1993

NEWCASTLE UNIVERSITY LIBRARY

093 50839 4

71.1.1.13121

Abstract

In the 1980s, multiple-processor computers (multiprocessors) based on conventional processing elements emerged as a popular solution to the continuing demand for ever-greater computing power. These machines offer a general-purpose parallel processing platform on which the size of program units which can be efficiently executed in parallel - the "grain size" - is smaller than that offered by distributed computing environments, though greater than that of some more specialised architectures. However, programming to exploit this medium-grained parallelism remains difficult. Concurrent execution is inherently complex, yet there is a lack of programming tools to support parallel programming activities such as program design, implementation, debugging, performance tuning and so on.

In helping to manage complexity in sequential programming, visual tools have often been used to great effect, which suggests one approach towards the goal of making parallel programming less difficult.

This thesis examines the possibilities which the **dataflow** paradigm has to offer as the basis for a set of visual parallel programming tools, and presents a dataflow notation designed as a framework for medium-grained parallel programming. The implementation of this notation as a programming language is discussed, and its suitability for the medium-grained level is examined.

Acknowledgements/Dedication

Although this thesis and the research it describes are my own work, I would like to express my gratitude to several people whose support I received in the course of the work. First among these must come my supervisor, Professor Peter Lee, for encouragement and guidance throughout my time at Newcastle. Additionally, I would like to thank Dr. Graham Megson and Dan McCue for constructive criticism on many pieces of work which eventually contributed to this thesis.

I spent three months in Rennes, France during the course of my research, and I would like to thank Françoise André for her support and encouragement during that time.

Finally, I acknowledge the financial support of the Science and Engineering Research Council of Great Britain during the first three years of my research, and of the EC ERASMUS scheme for funding my stay in Rennes.

This work is dedicated to the memory of my father, Dr. Anthony John Harley, 1933-1987, at one time an honorary lecturer in computer science at this university.

Table of Contents

<i>List of Figures</i>	7
Chapter 1 Introduction	9
1.1 Parallel Computing	9
1.2 Parallel Programming - the Problems	11
1.3 Solutions to the Problems	14
1.4 Dataflow	17
1.5 Main Contribution	20
1.6 Structure of the Subsequent Chapters	22
Chapter 2 Parallel Programming	23
2.1 Multiprocessors	23
2.1.1 The von Neumann machine	24
2.1.2 SISD Architectures	25
2.1.3 SIMD Architectures	25
2.1.4 MIMD Architectures	26
2.1.5 Distributed-memory Architectures	27
2.1.6 Shared-memory Architectures	29
2.1.7 Grain Size	30
2.2 Parallel Programming Models	32
2.2.1 Types of Parallelism	33
2.2.2 Parallel programming models	34
2.3 Programming Tools	38
2.3.1 Tools for Parallel Programming	39
2.3.2 Visual Programming	40
2.4 Dataflow	44
2.4.1 Dataflow for Parallelism	44
2.4.2 Architectures	46
2.4.3 Low-level dataflow languages	48

2.4.4	Dataflow design languages	49
2.5	Summary	51
<i>Chapter 3</i>	The MeDaL Notation	53
3.1	Concepts	54
3.2	MeDaL Semantics and Syntax	56
3.2.1	Datapaths	56
3.2.2	Actors	58
3.2.3	Companies	62
3.3	Design Decisions	63
3.3.1	General Visual Syntax	64
3.3.2	Actor Semantics	65
3.3.3	Datapath Semantics	67
3.3.4	Persistent Memory	69
3.3.5	Multiple Streams	72
3.4	Examples	79
3.4.1	The MeDaL Library	79
3.4.2	Example MeDaL Program	81
3.5	Summary	85
<i>Chapter 4</i>	Implementation Issues	87
4.1	Functions of the Programming System	88
4.1.1	The Division of Work	88
4.1.2	The Method Code Transformer	90
4.1.3	The Harness Generator	91
4.1.4	The Run-time System	92
4.2	Programming Language Interface	93
4.2.1	Programming actors in C and FORTRAN	94
4.2.2	Programming Interface in C++	95
4.3	Distributed-memory Run-time System	99
4.3.1	Datapaths Between Distributed Actors	99
4.3.2	Run-time System for Distributed Actors	101

4.3.3	The Wrappers of Distributed Actors	103
4.4	Shared-memory Run-time System	105
4.4.1	Shared-memory Datapaths	106
4.4.2	Shared-memory Wrappers	107
4.4.3	Other Functions	110
4.5	Summary	111
 <i>Chapter 5</i>	 Performance Evaluation	 113
5.1	Implementation Environment	115
5.2	MeDaL Run-time System Implementation	117
5.2.1	General Configuration	117
5.2.2	Specific Functions	118
5.3	Experimental Results	121
5.3.1	Basic Functions	122
5.3.2	Use of Basic Functions within Programs	123
5.3.3	Varying Numbers of Actors	129
5.4	Evaluation of Results	135
5.4.1	Grain Size	136
5.4.2	Limitations	137
5.4.3	Conclusions	138
 <i>Chapter 6</i>	 Conclusions	 141
6.1	Summary	141
6.2	Conclusions	145
6.3	Future Work	149
6.4	Closing Remarks	151
 <i>Bibliography</i>		 153
 <i>Appendix A</i>	 MeDaL Classes	 164
 <i>Appendix B</i>	 Example Application	 172

List of Figures

Figure		Page
1.4a	<i>Dataflow graph to calculate $x=4y + \sqrt{z \div 7}$</i>	18
2.1a	<i>Examples of interconnection in distributed-memory multiprocessor architectures</i>	28
2.1b	<i>Typical shared-memory architecture</i>	29
2.1c	<i>Definition of multiprocessor grain sizes</i>	31
3.2a	<i>Datapath syntax</i>	56
3.2b	<i>General-purpose actors</i>	60
3.2c	<i>Source actor</i>	60
3.2d	<i>Sink actor</i>	61
3.2e	<i>Merge actor</i>	61
3.2f	<i>Replicator</i>	62
3.2g	<i>Company (component view)</i>	63
3.3a	<i>Memory entity</i>	70
3.3b	<i>Persistent memory within an actor</i>	71
3.3c	<i>Output looped back to input</i>	73
3.3d	<i>Equalising method merge</i>	74
3.3e	<i>Datapaths to persistent memory</i>	75
3.3f	<i>Shared memory</i>	76
3.3g	<i>Demand-driven dataflow</i>	77
3.3h	<i>Synchronous and Asynchronous inputs</i>	78
3.4a	<i>Standard input and output actors</i>	80
3.4b	<i>File input and output actors</i>	80
3.4c	<i>Halt actor</i>	81
3.4d	<i>MeDaL example program "matrix-mult"</i>	82
4.0a	<i>Modules of a MeDaL programming system</i>	87
4.2a	<i>Wrapper and real method functions using C++ interface</i>	98
4.4a	<i>Shared-memory transmit algorithm</i>	109
5.3a	<i>Basic RTS functions</i>	122
5.3b	<i>Time-activity diagram - scenario 1</i>	125
5.3c	<i>MeDaL diagram - scenario 1</i>	126

5.3d	<i>Timings - scenario 1</i>	127
5.3e	<i>Time-activity diagram - scenario 2</i>	127
5.3f	<i>MeDaL diagram - scenario 2</i>	128
5.3g	<i>Timings - scenario 2</i>	128
5.3h	<i>Simplified matrix multiplication example</i>	130
5.3i	<i>Matrix multiplication - speedup against sequential MeDaL program</i>	132
5.3j	<i>Matrix multiplication - speedup against serial program</i>	134

Chapter 1

Introduction

1.1 Parallel Computing

The relentless drive for higher throughput in manufacturing industries has led factory managers not only to buy faster machines, which can speed up their production line by producing parts faster, but also to seek greater efficiency on the factory floor - for instance, by having two parts which are not components of each other made by different machines at the same time.

Such gains in throughput are also desired from computing equipment; there is a clear and continuing demand for computers which can do more processing in less time. Not only are computers continually developed with ever faster number-crunching power, but the concept of increasing computational capacity by executing mutually independent computations in *parallel* is also increasingly being employed.

In the race to build ever more powerful computers, the designers of the leading edge machines have therefore chosen paths which are leading away from the sequential processing model of John von Neumann, into designs which allow the concurrent processing of a number of parts of a computation. Of course, the success of these techniques relies on parallelism being available for exploitation in the software being developed. It is not always possible to parallelise: in the general case, any two instructions can be executed concurrently only when the data operated on by each one is not a direct result of the other - in which case there is no direct **data dependency** between those instructions. The same can be said of program modules which have inputs and outputs: two modules can only be executed concurrently when none of the inputs of either is an output of the other. This is equivalent to two machines being able to make parts of a man-made product which are not components of one another.

For some computer applications, the absence of data dependencies between instructions is obvious; an example is vector multiplication, in which the

multiplication of each element of the vector by a scalar does not depend on the value of any of the vector's other elements. The usefulness of this characteristic of vector operations has led to one type of architecture for computers known as vector processors, which exploit the parallelism of vector operations by processing elements of a vector concurrently.

However, such vector-based machines are clearly specialised in terms of what they are good at. Many, if not most, commercial applications do not employ large amounts of vector processing which could be executed faster by employing the type of machine just described. More general-purpose parallel machines have been built employing a number of (general-purpose) von Neumann processing units, each of which may execute different instructions on different data, in parallel. These machines are classified as MIMD machines [Flynn66], because they process Multiple Instruction streams and Multiple Data streams (as opposed to conventional, uniprocessor von Neumann machines which would be classified as SISD). Of course, from time to time, data dependencies arise between the streams of execution on different processors, so the separate streams need to communicate. In most such machines, this communication is initiated by the software.

So, unlike some of the vector processors, on most **MIMD multiprocessors** the control of parallelism is the responsibility of the software, not the hardware. The advantage of this is that such computers are general-purpose; just as a factory manager may often prefer a versatile machine to a specialised one, MIMD multiprocessors appeal to those who may wish to exploit parallelism in many different types of application. The disadvantage of controlling parallelism in software is that decoding this software into hardware instructions, and executing them, takes valuable time.

On machines in which the control of software is driven by parallelism, a parallel application must ultimately be expressed in the form of a parallel program with explicitly parallel sections, and explicit communication between these sections (though this form of the program is not necessarily visible to the programmer). Such is the variety of architectures even within the family of MIMD multiprocessors that this communication between sections of a parallel program may take one of several forms. One way of classifying communication types is by categorising architectures as **shared memory**, in which the multiple processing units share the

main memory, or **distributed memory** in which each processing unit has some local memory which the others cannot access. Communication between processors is required when data produced by a process executing on one processor is needed by a process running on another: in the shared memory case, communication may take place through some synchronisation mechanism such as joint access to "flags" in memory which indicate when the data in question is ready, while with distributed memory, communication is through "message passing" between processing units (the messages being the data required). However, even within these two basic architectures, there are many variations each with different strengths and weaknesses, which therefore require different programming techniques in order to be exploited efficiently. It should be re-iterated that on MIMD multiprocessors, the communication between processes within an application is under software control - so the onus of controlling communication is generally on the programmer. This may be the programmer of the application, or when the program is written in a high-level language (**HLL**) it may be the programmer of the HLL compiler who is ultimately responsible for generating the parallel program.

1.2 Parallel Programming - the Problems

Just as a commercial industrial product such as a turbine generator has a "life-cycle," consisting of design, manufacture, testing, use, and maintenance, so does a software product. The first three phases of the software life-cycle (design, implementation and testing) can be loosely termed "programming". The term **parallel programming** refers to the process of making an explicitly parallel program, from the design stage onwards. This definition excludes the use of those HLL compilers mentioned at the end of the previous section which generate parallel code from programs which are not explicitly parallel. Although such compilers can be a cost-effective way of parallelising existing programs, the code they produce is usually not as efficient as that of programs designed from scratch to run in parallel, since such compilers can only detect potential parallelism in the *implementation* of a program, and not parallelism in the *design*. Thus, the interest in parallel programming derives from the desire for more efficient parallel programs.

However, it is widely accepted that parallel programming is inherently difficult compared to traditional sequential programming. Apart from the fact that it is a relatively new and immature field, there are three main reasons for this difficulty:

- the complexity of managing parallelism from the design stage onwards;
- the low-level nature of many parallel programming languages;
- the lack of debugging support.

The problem of managing the complexity of parallel programming occurs mainly at the design stage if the program is designed from scratch (if not, it can occur later). In some applications the potential for parallelism may be obvious, such as in ray-tracing (the generation of images by tracing rays of light as they reflect off representations of objects), where the final value of any pixel depends only on the given representation of the scene, and not on the final value of any other pixel. In such cases the input data and work are simply divided into pieces, which are shared out among the available processors. However, in other applications - in which the data dependencies are far more complex - it is much less obvious where parallelism can be efficiently exploited. On one hand, if the programmer attempts to parallelise two computations between which there is a dependency, one computation may spend time waiting for the other, which is inefficient. On the other hand, if the programmer plays safe and places the two computations in sequence, computations which could have been executed concurrently are not, and again the program is less than optimally efficient.

The programming difficulty described above is important because multiprocessors are generally employed in situations where speed (and hence, maximum efficiency) are considered of paramount importance, since that was the motivation for developing multiprocessor technology; yet most existing programming languages (explicitly parallel or not) concentrate on providing support only for implementation, not for parallel program design, or software maintenance at the design level.

After the problems of design complexity have been dealt with, the next problem which must be faced is that of the low-level nature of parallel programming. The terms low-level and high-level refer to proximity to the machine's view of a program (hardware instructions) and to the user's view (mathematical or real-world objects to be manipulated) respectively. A higher "level" allows the human

programmer to understand code better, and hence be more productive. This principle led, in the 1950s and 1960s, to the development of what were at the time termed "high level languages," a term which has already been introduced in this chapter. However, the reality is that although these HLLs were higher-level than machine code, they fell far short of being a natural way of expressing programs; and subsequently languages have been developed (for sequential architectures) which offer a higher level still, and these will be discussed in the next section.

Meanwhile, the multiplicity of parallel architectures described in section 1.1 has led to a proliferation of "parallel" versions of the older, sequential, textual HLLs. Many of the programming languages supplied by multiprocessor manufacturers are simply extended versions of popular languages like C and FORTRAN. It is a characteristic of these parallel languages that they typically reflect the target architecture in their language constructs, in other words, whether it is message-passing or shared memory. Thus, the programmer must expend time and effort on controlling the machine (simply to make the program work in parallel) rather than on making the application more efficient.

Of course, this reliance on architecture-specific constructs makes the programs difficult to "port" between different machines as well as laborious to write; and the focus on the mechanics of parallelism only aggravates the difficulties of identifying and hence implementing any potential for parallelism in the application being developed (a problem inherited from the design phase due to the lack of design tools, as identified above).

Moreover, the need for low-level machine knowledge has kept parallel programming a domain for specialists, those who possess detailed knowledge about the architecture and operating system of their parallel machines. Such a situation has long been recognised as a cause of much expense and delay in producing and maintaining software for multiprocessor computers, and is entirely due to this low-level bias of current parallel programming languages.

In the third phase of parallel programming - testing - problems arise because there is little support for the debugging of parallel programs. Debugging tools for parallel systems are in their infancy, and tend to concentrate on low-level, architecture-specific aspects of the program such as communication, echoing the problems already outlined in this section. Parallel programs can fail for many

reasons which are not a problem in sequential programming, for instance because of synchronisation problems between co-operating **tasks** (sections of a program which carry out a particular function, such as multiplying a matrix or generating a table of results) or side-effects between particular combinations of tasks if they happen to be executing concurrently. Not only do few tools adequately address these problems, but where timing problems are involved, the presence or absence of monitoring for debugging purposes can be "intrusive" and affect the behaviour of the program.

Even when debugging tools are available, they are often in the form of a "toolkit", a collection of different tools to aid different aspects of programming, with little or no integration, or even uniformity of user interface; each tool providing its own model of how the parallel program works (examples of tools being profilers and symbolic debuggers). Reconciling these models, understanding all the different possible views, and switching between them, is a further burden for the parallel programmer. When bugs require changes to the implementation or even the design of the program, programmers must return to the tools used for these earlier phases, increasing the number of tools which they must deal with, and hence the difficulty of programming.

These problems in the design, implementation and testing of parallel programs clearly illustrate that the problems of parallel programming extend across several phases of the software life-cycle. Little effort has yet been put into developing Computer-Aided Software Engineering (CASE) tools for supporting the design of parallel programs and carrying this design through to implementation, let alone debugging. There is no widely accepted "methodology" for the process of building parallel software. In the 1970s, the lack of this kind of advanced software support for sequential programming led to the perception of the "software crisis". A similar state of affairs still applies in the field of parallel programming.

1.3 Solutions to the Problems

The field of sequential programming, being somewhat older than that of parallel programming, is therefore more mature. In considering solutions to the three-sided problem described above, it is useful to look at the ways in which software support for sequential programming has developed.

The difficulties arising from having to control computers at too low a level has been recognised for decades, and the ready answer to this is **abstraction**. Abstraction is what makes the HLLs discussed earlier "higher-level" than machine code; each sequential HLL implements a "virtual machine" which, abstracting away from the underlying (von Neumann) architecture, hides technical details from the programmers, enabling them to concentrate on the correctness of their algorithms. The more abstraction is applied, the higher-level a language becomes; the first level of abstraction produced assembly languages, and the second, HLLs such as FORTRAN, PL/I, PASCAL and C. Because this second level of abstraction hides machine-language details, software implemented in these languages is easily "portable" to machines with different machine-languages.

In the 1970s and 1980s, abstraction was taken further and languages were developed which were even further from the underlying machine - sometimes adopting visual representations to aid understanding (examples of these "third-generation languages" are Ada, with its Mascot diagrams, and Smalltalk). However, it should be noted that with higher-level languages there is usually a trade-off between ease of understanding and efficiency; at a lower level, programmers have greater control over the machine, and hence can "tune" their code for greater efficiency (at the expense of potentially making more mistakes). Given that a slow, correct program is usually preferred by customers to a fast, incorrect one, the fact that higher-level languages have not been more widely adopted than they have can only be put down to a great optimism in the ability of programmers to produce correct code.

So far, though, third-generation languages have mostly been restricted to sequential architectures. This is probably due to several factors: firstly, the time-lag between sequential and parallel computer technology. Secondly, the lack of a clear winner so far in the search for parallel programming models for parallel languages to be based on. And finally, perhaps, the fact that multiprocessors tend to be used in situations where speed is of the essence (as mentioned above), and hence efficiency is considered more important than the amount of programmer effort that will be needed to produce correct code.

In the meantime, the level of abstraction represented by second-generation HLLs (especially C and FORTRAN) has become more or less standard in parallel

programming. Certainly, in adopting a familiar form of programming one reaps a benefit from the familiarity most programmers will have with it. However, there are problems with adapting this type of programming language for parallelism. The most obvious arises from the inherently sequential nature of existing HLLs; HLL programs take the form (on paper or on the screen) of a linear array of textual statements. This clearly does not accurately model the behaviour of even a sequential program (because of control jumps, loops, calls subroutines and returns, etc.). For describing the concurrent sections of a parallel program, textual languages are even more inadequate. So, although programmers do not have to map algorithms onto low-level machine operations, they do have the burden of mentally mapping a sequential list of statements into a parallel form.

The difficulty of this mapping is clearly linked to the first aspect of the parallel programming problem, namely the complexity of parallelism. Complex parallel programs can contain a web of data dependencies which are by no means easy to understand; and when one has to derive a mental model of a parallel system from a low-level textual description such as a HLL as described above - as is necessary in software maintenance - the problem is even more acute.

One solution to this lies in the provision of more sophisticated design tools. For instance, cognitive science reveals that humans can interpret and understand diagrams much more easily than text, and the developing field of "visual programming" attempts to put this principle into practice. Visual programming environments have been developed for uniprocessor computers, replacing the traditional textual programming language with a **visual language** which can be entered and manipulated on a graphical display. The freedom which visual programming offers from linear arrays of text should be especially useful in parallel programming, with its additional complexities of concurrency and data dependencies. The benefits of visual programming environments in terms of helping programmers to manage complexity are becoming increasingly widely recognised: for instance, the appearance of Smalltalk, with its manipulation of graphical representations of objects, revolutionised the field of object-oriented programming. However, there has been no analogous contender in the parallel programming arena.

The application of visual programming techniques to parallel programming has

implications on the other aspects of the parallel programming problem as well. In a visual programming environment, it is *visual abstractions* of lower-level functions and/or data which are manipulated, bringing the greater ease of interpretation that diagrams have over text. Also, a visual programming environment provides a ready framework into which further programming tools, such as debugging aids, can be integrated. Thus the benefits of abstraction and visual programming can be brought to the task of debugging parallel software.

To summarise, it seems that an obvious strategy to help alleviate the problems of parallel programming would be to provide a visual programming environment. In such an environment the design, implementation and debugging of parallel programs could take place, entirely at a high level, employing visual abstraction. The question therefore arises, what would be a suitable visual abstraction for such a system to be based upon? One answer to this question is a visual abstraction known as the dataflow graph, also called the dataflow schema.

1.4 Dataflow

To explain what a dataflow graph is, it is first necessary to consider the concept of dataflow as opposed to control flow.

Algorithms are almost always programmed on sequential computers in a form which describes the flow of control through the program, in terms of sequences of instructions and jumps to other sequences. Data items are defined in terms of the program's operations, for instance as operands to instructions and as parameters to be passed when a subroutine is called. The class of HLLs described as imperative languages implement this paradigm directly, using constructs such as loops, procedures and so on, which are abstractions (hence the name High Level) of machine operations. The other classes of HLLs, such as functional languages, translate other language forms into control flow form. This focus on control flow merely reflects the fact that all such languages are programming a virtual machine which is based on the von Neumann model. However, for parallel machines there is no one accepted virtual machine which HLL designers can agree on; no one **bridging model** to bridge the gap between radically different architectures and one software programming environment. This is one reason why new models must be explored.

One model to emerge is the **dataflow** model, so called because its focus is not on the flow of control through a program but on the flow of data. Operations are defined in terms of the data they operate on, and not vice versa as in control flow. Thus, while a control flow program specifies the order in which operations must be executed, a dataflow program consists of a collection of operations together with a description of the way in which data flows between them; first the data is transformed by one operation, then another, and so on.

Programs of this form can easily be visualised as a directed graph, with data flowing along the arcs and being processed at nodes which represent the operations. A very simple dataflow graph is illustrated below.

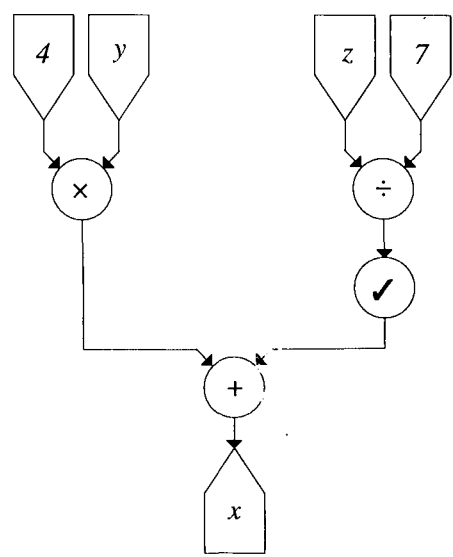


Figure 1.4a: Dataflow graph to calculate $x = 4y + \sqrt{z \div 7}$

In this simplified graph, data (in this case, numeric variables and constants) flow along explicit paths described by the lines, through **operations** (simple mathematical functions) represented by circles. The other shapes represent the input to and output from the program. Each operation has one or more inputs and one or more outputs, and can be executed when all of its inputs have become available.

The interest in the dataflow model with relation to parallel programming is made clear by examining a dataflow graph. Since all data dependencies are explicitly shown in the graph, it is implicit which operations do not depend upon each other for data, and can therefore be executed in parallel. In the example above, the multiply and divide operations can always be executed in parallel, but the

addition operation must wait until both the multiply and the square-root operations have produced a result. Furthermore, if multiple instances of x are to be calculated from multiple pairs of y and z , the division operation can execute on one value of z while the square-root operation is still executing using the division's previous result; a pipeline can be set up to increase throughput, in a way exactly analogous to a factory assembly line. In terms of data dependencies, this is possible because the division does not depend on the result of the square root; there is no data path from the square root to the division operation. This relationship between two such unconnected operations is known as **implicit parallelism** because it is implied by (implicit upon) the explicit data dependencies. Because operations can operate only on rigidly defined sets of data, namely their own input paths, there can be none of the "side effects" (one operation interfering with another's data) which have long been recognised as a problem in software engineering.

The value of dataflow graphs as a design tool has long been recognised. Programs can be designed in the form of such graphs, and then implemented using the insight into the data dependencies gained from reading the graph. Since the graph is an abstraction, the design is not architecture-specific, and because it provides a visual overview, it should in general make the program easier to understand. Some systems analysis methodologies do indeed make use of data flow diagrams. The data flow design is usually a stage before a lower-level design of the program in the form of (control) flowcharts. It is a natural step to design programs in a control flow form if the programming is to be done using a HLL based on the sequential control flow model.

However, it has also been established that it is possible to implement programs directly from these familiar data flow descriptions, or at least from a modified form of them. A **dataflow program** consists of the same type of operations as a traditional program, but instead of having a predetermined sequential order of execution, each operation is executed at any time after the resources it needs have become available: the data it operates on, and a processor to execute it. Dataflow programs are programs for a "virtual machine" based on the dataflow model. The model does not assume any particular number of processors, but clearly the more processors available, the more potential parallelism in a program can be exploited.

The term "operations" is deliberately vague, since there are several possibilities

for its definition. An operation could be an individual machine instruction, a definition which typically results in a very large amount of parallelism in a program. As there is an overhead associated with every instruction (that of deciding when its resources have become available) this form of dataflow becomes very inefficient on conventional computers. Specialised **dataflow machines** are currently under development which seek to deal with these overheads in hardware, a subject which is discussed further in Chapter 2.

An alternative definition is **Large Grain** dataflow in which the operations are independent program modules which are themselves based on control flow; they exchange data between separate Von Neumann processors, which may even be located in different machines. The term **grain size** refers to the size of units which may be executed in parallel. Large Grain dataflow is also discussed in Chapter 2.

While fine-grained dataflow requires specialised architectural features, and Large Grain Dataflow was designed with loosely-coupled networks of processors in mind, the MIMD multiprocessors discussed in the introduction operate efficiently at an intermediate grain size. It is this combination of medium-grained parallelism and dataflow techniques which this thesis examines.

1.5 Main Contribution

It was stated earlier that dataflow graphs are a useful design tool. However, given software support, there is no reason why dataflow programs should not be directly compiled into executable code, providing an alternative to the textual HLLs currently in use.

The control-flow HLLs used to program MIMD multiprocessors usually employ their function or procedure constructs as the units which are executed in parallel. This thesis proposes a dataflow system in which this level of parallelism is used: the "operations" in a dataflow graph are written in a familiar imperative HLL. In fact, the dataflow system could act as a **harness** around already-written functions of a sequential program, as a means of parallelising it.

Using such a system, the programmer would be able to design a program in the form of a dataflow graph on a visual display, using a graph editing tool. The operations are "filled in" in windows using a traditional text editor; and the system

uses the graph and the code segments to generate a parallel program, extracting the implicit parallelism from the dataflow graph. This is analogous to the HLL compilers discussed earlier which produce parallel code from sequential; but the software development system proposed here has the benefit of access to the program's design (the top level of abstraction) and not just implementation code. Using the editors, it would be possible to modify the graph and code sections at will, and visual aids to debugging could be presented in terms of the dataflow graph - in other words, the whole parallel programming process could take place at a high level of abstraction. This thesis contends that such a system is a viable solution to the problems involved with parallel programming described in section 1.2.

The system described uses a new form of dataflow graph notation specially designed for the purpose of supporting high-level visual programming of conventional (shared and distributed memory) MIMD multiprocessors. The features of this new notation are described fully in subsequent chapters.

In the field of parallel processing, the efficiency with which parallel programs execute is often considered of paramount importance, since faster execution is of course the main motivation for employing parallelism. Thus, the most important criterion for the success of the parallel programming system presented here is likely to be the efficiency of the executable parallel code which it generates. Therefore, the practical focus of this thesis is on the efficient implementation of a run-time system which enables segments of sequential code (as described above) to run in a dataflow fashion on a conventional MIMD multiprocessor architecture. The examination of the efficiency of an actual implementation of such a run-time system (on a shared-memory multiprocessor) forms the main practical result arising from this work.

Finally, the usefulness of such a system is evaluated, taking into account the conclusions on the efficiency of the dataflow approach, and the various possibilities the proposed system may hold for further software tools.

1.6 Structure of the Subsequent Chapters

The remainder of this thesis is structured as follows. The context of this work is described in Chapter 2: each of the main components is considered in turn (MIMD architectures, software tools for parallel programming and debugging, visual programming and the dataflow model) in terms of the previous research in each of these fields. The various ways in which the dataflow paradigm has been developed are summarised, relating some of the important concepts to the system to be considered in later chapters.

Chapter 3 gives details of the proposed notation for medium-grained dataflow, with the emphasis on its uses as a framework for parallel programming. The syntax and semantics of the notation are presented, with a worked example. There is a discussion of the main design decisions involved, focussing in particular on the provision of persistent memory and synchronisation mechanisms, as well as some of the alternatives which were considered.

In Chapter 4 the possibilities for implementations of the proposed system, on both shared-memory and distributed-memory architectures, are described. In particular, it is demonstrated how dataflow graphs can be automatically compiled into executable parallel code, forming a harness for HLL functions. The run-time support needed by the executable code thus generated is also described in detail.

Chapter 5 describes an actual implementation of a run-time system, as outlined in Chapter 4, on one hardware platform. Emphasis is given to demonstrating that the implementation allows the efficient execution of parallel programs in a dataflow fashion at the medium-grained level of parallelism.

Finally, Chapter 6 summarises what conclusions may be drawn from the experiences gained with the system, and what significance this type of system may have, within the context provided by Chapter 2. In particular, conclusions are drawn on what the potential range of application areas might be for this type of system. Possible future developments for the system so far developed, and potential related topics for research, are also outlined.

Parallel Programming

Having briefly described the parallel programming problem, and outlined the approach which this thesis proposes in order to move towards a solution, it is necessary to examine the components of the programming problem in more detail. Much research has already been undertaken into the problem of how to increase processing throughput by using parallelism, and to summarise all of this research would be a gargantuan task. Therefore, the discussion presented here is deliberately limited to the work which has had a direct input on the material developed in the remaining chapters.

Since the scope of this research is itself limited - to the study of "dataflow development of medium-grained parallel software" - this chapter serves to define the terms used in this title, and to present the important concepts involved in understanding these terms. The motivation for this research is to examine the possibilities for the provision of a graphical software development environment for medium-grained parallel software; this chapter focuses on four key areas - the underlying medium-grained multiprocessor technology; parallel programming models; parallel programming tools, including visual tools; and dataflow. An understanding of each of these facets of the parallel programming problem is necessary in order to understand the original work presented in subsequent chapters.

2.1 Multiprocessors

Much research into parallel programming has come about as a result of the availability of parallel computers, the architectures of which allow multiple operations to occur concurrently. Since programming consists of controlling what operations take place, it is necessary to examine the technology underlying parallel programming.

2.1.1 The von Neumann machine

Multiprocessors are descended from single-processor technology, so before considering multiprocessors, it is useful to review the important concepts of uniprocessing. For many years, most uniprocessors have been based on the "von Neumann machine" [vonN58]. A simple von Neumann machine consists of a memory **store** which contains instructions and data, and a **processing element** which performs operations in a perpetual cycle of fetching a coded instruction from the store, decoding and executing it, the result being put back in the store (the fetch-decode-execute cycle). One element of the store is designated as the **program counter** - a pointer to the next instruction to be fetched - and this is automatically updated at the end of each cycle. The sequence of operations can be selected by the program itself, by altering the program counter.

In the general case, each instruction has at least one operand (input) and a result (output). As stated in Chapter 1, in principle when the output of any two instructions is not the input of the other, those two instructions can *potentially* be executed in parallel. However, the existence of only one processing element and one store means that in practice, no two instructions can be executed concurrently, and this has led to experimentation with other architectures which derive from the simple von Neumann model.

In fact, a wide variety of architectures have been experimented with, working on a variety of principles, and so a need has arisen for a classification scheme to organise these computer architectures into "families" governed by the same operating principles. One such scheme has been proposed by Flynn [Flynn66]. Although Flynn's classification is fairly limited in terms of today's architectures, and various enhancements have subsequently been proposed (for example [Dunc90]), Flynn's scheme remains a useful and elegant way of describing broad types of architectures. Flynn's classification is simply based on the **instruction streams** (how many instructions are fetched concurrently) and **data streams** (how many items of data are fetched concurrently to be operated on) in an architecture. These two characteristics are simply classified as "single" or "multiple" streams. Thus, in Flynn's scheme there are four basic families of architecture: SISD (Single Instruction stream, Single Data stream), SIMD (Single Instruction, Multiple Data), MISD (Multiple Instruction, Single Data) and MIMD (Multiple Instruction,

Multiple Data). There are many examples of each of these types of architecture, with the exception of MISD machines, which has not, so far, proved to be a useful configuration.

2.1.2 SISD Architectures

The conventional Von Neumann machine is of course classed as SISD, since it has fetches a single instruction to operate on a single data item in each cycle. It is worth noting that a certain amount of parallelism has been introduced into SISD architectures, particularly employing an instruction **pipeline** in which as one instruction is being executed, the next is decoded, while the following instruction is being fetched, all concurrently. Although this is a very limited form of parallelism, it is an important concept and is used in many of the other architectures described below, in addition to their other forms of parallelism.

2.1.3 SIMD Architectures

SIMD machines, as their name suggests, execute the same instruction on multiple data items concurrently. There are a number of different architectures in this class, among which are array and vector processors. Array processors typically have a single control unit, and an array of directly connected processing units; the processing units each have their own registers and store, and hence operate on different data items, but the execution of operations is directed by the control unit (a **master-slave** arrangement). An early example of this type of architecture was the Illiac-IV [Barn68], a more recent example being the DAP [Hock81]. Vector processors are similar, achieving parallelism by providing hardware which allows whole vectors to be used as operands to instructions, and producing whole vectors as the result, the same (single) instruction being applied to each element of the vector concurrently. The TI-ASC [Wats72] and the Cray-1 [Russ78] are typical examples of vector processing architectures. To achieve further parallelism, some vector processors employ memory pipelining (referring to the fetching of vectors from store) in addition to the instruction pipelining described above.

Although SIMD architectures have been very successfully employed in many heavily numerical applications, these architectures have their limitations in terms of applicability. Any master-slave arrangement is liable to be subject to bottlenecks in some cases; in the case of array processors, when there is insufficient data

parallelism - that is, when there is less data to be operated on than there are processing units - the full speedup cannot be achieved. A problem when using vector processors is that the vectors used for operands and results are of a fixed size on each machine, and when a problem requires vectors which are, for example, somewhat greater than this size, there is the same problem of inefficiency due to a mismatch of data parallelism. So, although SIMD architectures can often be used very successfully if used for applications which map well onto their particular architecture, they lack the generality to be applied to a variety of problems - such as those containing vectors of varying sizes, or indeed the many computing tasks containing no significant amount of vectorisable mathematics.

2.1.4 MIMD Architectures

Since MIMD machines can concurrently execute different instructions on different data, this gives them greater flexibility of control than SIMD machines. This makes them a more general-purpose solution to parallel programming. MIMD machines have already been employed in many different application areas. As with the SIMD family, there are a number of different architectures within this classification; among the approaches to MIMD multiprocessing which have been tried, are:

- Systolic arrays [Kung82], which have an array of interconnected processing elements which operate on data pumped through the array. Various topologies for the arrays are appropriate for different applications; in the CHiP [Snyd82] the array topology is reconfigurable, providing a general-purpose machine.
- Dataflow architectures, which are described in more detail in section 2.4.
- Shared-memory architectures, in which a number of processors share a store, usually through a common bus.
- Distributed-memory architectures, containing a network of communicating processors each with its own store.


Of these, the latter two have become by far the most prevalent. Their strength lies in the fact that the processing units involved in these architectures are, in themselves, essentially von Neumann machines; multiprocessors based on these architectures usually incorporate a number of commonly-available uniprocessor processing elements linked together. This strategy is particularly cost effective,

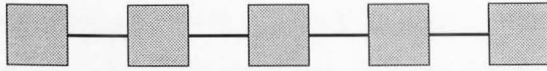
since it "rides on the back of" advances in uniprocessor technology, and only the hardware technology which needs to be developed specifically for these multiprocessors is that of the communication links between the processors (and protocols for using them efficiently).

There is, though, a trade-off between speed and generality; after all, hardware solutions to problems are in general faster than software solutions, and the more general-purpose machines lack the specialised hardware of, for instance, vector processors. The fact that on conventional multiprocessors parallelism is managed by software, leads to the variety of parallel programming models described in section 2.2. However, since many of these models are based on the underlying architecture involved, it is first necessary to examine these architectures in more detail.

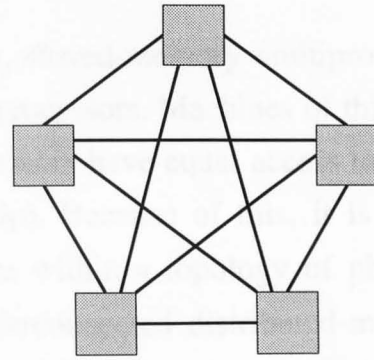
2.1.5 Distributed-memory Architectures

Distributed-memory multiprocessor architectures consist of a number of processing **nodes**, with each node comprising one processor and a memory store which is only accessible by that processor; there is no globally accessible memory. Processors communicate by sending data **messages** to other nodes via communication links, each processor being linked to one or more other nodes. However, the more nodes each node is linked to, the greater the sophistication of communications electronics needed, which limits the number of links implemented in practice. Architectures exist with a number of different topologies of interconnection between the nodes, of varying complexity from linear arrays, to hypercubes (notably the Intel iPSC [Gehr88]), to complete connectivity. Figure 2.1a illustrates the first and third of these arrangements, though it should be emphasised that there are many others.

 = 1 node (memory + processor)



Linear Array



Fully interconnected

Figure 2.1a: Examples of interconnection in distributed-memory multiprocessor architectures.

An additional limitation is that the bandwidth of inter-node communications is usually much lower than that between a node's processor and its local memory. This means that algorithms must be found with a minimal amount of communication for maximal efficiency. Indeed, in general it is preferable to undertake extra processing if this can avoid extra communication [Padd93]. Naturally, if not all processors are connected to each other, the cost of communicating between any two nodes increases in proportion to the number of "hops" messages must make. From the programmer's point of view a completely-connected architecture is the simplest to use, but on any other type of distributed-memory multiprocessor, programming involves finding an algorithm which is efficient on the available topology of processors. This makes the development of software for these machines far more complex than that of software for uniprocessors.

Nevertheless, distributed-memory multiprocessors have a number of attractive features which have made them popular; perhaps the most important of these is scalability. This arises because each inter-node link can be a simple point-to-point communications bus, used only by the two processors involved. Therefore, further processors (and hence memory) can be added to the architecture with new inter-node links as appropriate, without placing a greater load on existing links. For applications in which the computing load is very large, and the communication needed between tasks is relatively small, this makes distributed-memory multiprocessors attractive.

2.1.6 Shared-memory Architectures

In contrast to distributed-memory multiprocessors, shared-memory multiprocessor architectures share memory between a number of processors. Machines of this type are typically symmetric, in the sense that all processors have equal access to all of the memory (there is no master-slave relationship). Because of this, it is never necessary for a particular process to know where within a topology of physical processors it is running, unlike any non-fully-interconnected distributed-memory multiprocessor. There may in fact be several physical banks of memory, but the mapping of logical to physical addresses is easily done by hardware; at the software level there appears to be only one logical memory. Because software does not need to know about physical processor or memory location, the programming model of these architectures is very simple: all data in memory is normally available immediately, the only basic access mechanism needed being the use of software **locks** (usually supported by hardware instructions) to restrict concurrent access, for instance in a concurrent producer/consumer situation.

Memory access is typically through a common bus (figure 2.1b). Because this medium is shared, bus bandwidth becomes the limiting factor in expanding the number of processors in the machine.

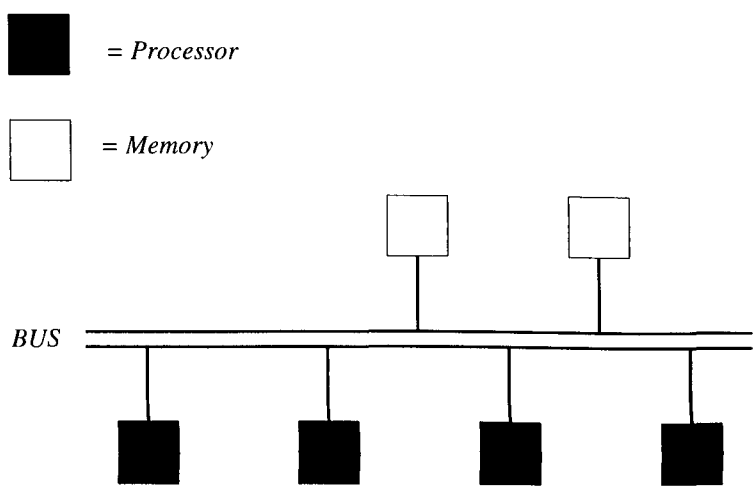


Figure 2.1b: Typical shared-memory architecture

Shared-memory machines have been built in a range of orders of magnitude of processing power, from the Firefly multiprocessor workstation [Thac87] to the 2 to 20 processor Encore Multimax [Enco89], to the Cray X-MP supercomputer

[Lars84]. Whatever the scale, the attractiveness of this type of architecture lies mainly in the simplicity of the programming model.

The distributed-memory and shared-memory models are not completely exclusive. A few hybrid architectures such as the KSR-1 [Kend92] have used elements of both models. Moreover, a current research topic is the implementation of a "virtual" shared memory on distributed memory architectures. In such architectures, extensions to the operating system of a distributed memory machine provide the programmer with a logical shared memory, which is in fact distributed across the available nodes. Essentially when a memory access is requested, the system checks which node that location is physically stored on, and if it is not the current node, retrieves it from another node using message-passing - thus relieving the programmer of the burden of explicit message-passing. The current interest in virtual shared memory perhaps demonstrates the superiority of shared memory as a programming model.

2.1.7 Grain Size

Having described the most popular platforms for parallel processing in terms of their architecture, there is one further aspect of hardware which has a bearing on parallel software development, and must therefore be considered: namely, the question of whether the cost of making a program parallel is justified by the processing time gains achieved. There is always a "cost" associated with running a program in parallel, in terms of time spent performing tasks necessary to allow the program to run in parallel. On distributed-memory architectures, the major cost involved is that of transferring code and data to the node on which they will be needed; on shared-memory machines, the system calls needed to create tasks, process control blocks, manage locks and so on for parallel processes are the most significant cost. These costs form an "overhead" of time lost which must be compared to the time gained by parallel execution. For example, if a program contains a loop which takes time $20t$ to execute, and is repeated 5 times, the program will not execute faster if each iteration of that loop is executed as a different process, unless the time taken to start each of these 5 processes (each containing one iteration) is less than $16t$ ($80t$ in total).

This start-up cost is not the only consideration in deciding whether (and how) to make a program parallel. Even if the sections of a program which can be

executed in parallel are long compared to the start-up overhead, separate parallel tasks often need to co-operate with each other, and they can only do this when both tasks are ready - in other words, they need to **synchronise**. This synchronisation event could be, for example, a producer process making data available for a consumer process. Synchronisation events, too, carry an overhead. Where synchronisation occurs for tasks to pass data between one another, this overhead includes any time taken to copy the data from one area of shared memory to another, or to transmit the data from one processing element to another.

It is the time taken by these two types of overhead, in relation to the total time taken by parallel sections between synchronisation events, which determine whether or not parallelisation is worthwhile in a particular case. Thinking of time as a "size", the term **grain size** is used to describe the time taken by a process between synchronisation events. For any given multiprocessor architecture, there is a certain minimum grain size, below which parallelisation is not efficient; namely, grains which take less time to execute than does the code which implements the synchronisation (including communication time). This will vary from one architecture to another, since the time taken to synchronise depends on the hardware operations required.

To be more architecture-independent, it is more useful to define grain size in terms of numbers of instructions rather than absolute time. Gordon Bell proposed a classification of parallel architectures according to grain size defining fine, medium, coarse and very coarse grained parallelism [Lee87]. This is adapted in Figure 2.1c in an updated form which more accurately represents present-day usage. The grain size definitions illustrated below are adopted in the remainder of this thesis.

Grain Size	Synchronization Interval (Instructions)	Construct for Parallelism
Fine	<20	Inter-instruction (hardware controlled)
Medium	20-2000	Inter-task (within a single program)
Coarse	2000-10,000	Multiprocessing of concurrent processes
Large	10,000 +	Distributed processing across local area network

Figure 2.1c: Definition of multiprocessor grain sizes

This concept of a minimum grain size, which must be employed for parallelism in software to produce any gains in terms of reducing execution time, is the final important way in which the architecture of conventional parallel machines affects the software which runs on them. The next component of parallel programming to be considered is a level above the hardware - the programming models on which languages for parallel programming are based.

2.2 Parallel Programming Models

Very little programming in the 1990s is undertaken at machine code level, because of the complexity of programs currently being developed. When programming in parallel, that complexity is increased so greatly that high-level programming is essential. This section discusses the programming models of parallelism which current high-level parallel programming languages for conventional MIMD multiprocessors are based upon. This section, and the next section (on parallel programming tools), focus on control-flow based, rather than dataflow based, programming models. Dataflow is considered in more detail in section 2.4. This chapter also focuses on the operational, rather than formal, aspects of parallel programming.

Although the need for structured programming languages for parallel programming was recognised long ago [Tane78], parallel programming languages developed in the early 1980s nevertheless evolved in an ad hoc manner. Often the programming model used by such languages simply added the parallel processing

primitives provided by one particular architecture to an existing language, with little attempt at generalisation. This approach sacrifices ease of use and portability to efficiency of implementation, though this is not surprising given the emphasis on speeding up execution times. More recently, a number of more general programming models which are not architecture-specific have been introduced; and their usefulness has been proven on a wider range of real applications. This answers Karp's criticism [Karp87] that the early parallel languages were not tested on realistic enough examples to prove their worth for general-purpose parallel programming.

The remainder of this section describes the major models of parallelism on which parallel programming systems have so far been based. First, however, it is useful to identify an abstraction of parallelism with which to compare these models.

2.2.1 Types of Parallelism

One abstraction of software parallelism has been proposed by Carriero and Gelernter [Carr88], who identify three basic types of parallelism which a parallel programming model may offer the programmer, namely:

- result parallelism
- activity parallelism
- structure parallelism.

Taking each of these in turn, **result parallelism** is the type of parallelism used when a task is broken down into multiple sub-tasks, each of which executes in parallel to produce a different part of the overall task's result. For instance, consider a parallel relational database program in which a table Z is required which is the intersection of two tables Y and V , Y being the union of two further tables W and X , and V being the projection of tuple t over table U . The calculation of $union(W, X) \rightarrow Y$ and that of $project(t, U) \rightarrow V$ can be executed in parallel, producing the two partial results V and Y concurrently.

Activity parallelism is an alternative to result parallelism, in which a number of identical tasks are created to work on the data needing to be processed, taking slices of the data until it is used up. An example of this is a matrix operation in which each row or column of the resulting matrix could be calculated independently, in parallel. If there were more columns or rows to be processed than

available processors, one task might be created per processor, and each process would take the data needed for an output column from a "pool" of input data, until all the result columns had been calculated. This type of parallelism is typical of master/worker process organisation.

Finally, **structure parallelism** can be exploited when a task has a number of identifiably separate stages involved in producing its result, and a **stream** of data which must go through these stages. In such a case, separate processes can execute each stage in parallel while the data is streamed through, in a software equivalent to the pipelines described in section 2.1.

These three orthogonal types of parallelism may of course all be employed within the same parallel program - if the programming model allows. In the following section, various programming models are discussed, with reference to the ease with which these three types of parallelism may be employed by the programmer.

2.2.2 Parallel programming models

Just as the types of parallelism which can be expressed in a parallel programming language can be classified, it is useful to similarly classify the models for parallel programming which exist. Most models are based on the concept of processes, which undertake the computation and are within themselves sequential, and differ only in the way in which these processes synchronise when data dependencies occur. One way of classifying the models is according to whether they assume the existence of shared resources to aid synchronisation, or instead rely on message-passing; the obvious analogy being with the two main types of multiprocessor described above. Within the classification of message-passing models, there are two sub-types; those models which assume asynchronous communication, and those which use synchronous message-passing.

This classification along similar lines to the classification of architectures does not necessarily mean that each such programming model is specific to one particular architecture, or even to one type of architecture; portability can be achieved by providing the features demanded by the model in software. For instance, a message-passing system could be run on a shared-memory machine, by passing the messages through an area of shared memory in a controlled way. The

classification is merely a convenient perspective from which to view the available models. Each of the three types of model (shared memory, asynchronous message-passing, and synchronous message-passing) can now be considered in turn.

An early shared-resource model for synchronisation is the **semaphore** [Dijk68]. Semaphores provide an abstraction of hardware mechanisms for synchronisation; the semaphore has a value, and two operations, P (which decreases the value) and V (which increases it). The value, however, is not allowed to drop below zero and if it reaches zero, any process which subsequently tries to use the P operation on it can be suspended until another process uses a V operation on it. Thus, semaphores are a shared resource which can be used to synchronise processes. However, they are a low-level mechanism for synchronisation, and most models support higher-level abstractions for synchronisation. A higher-level model which has proved popular is that of the **monitor** [Hoar74]. A monitor is an object consisting of the resource, such as shared memory, and the operations which access that resource; access is not allowed from outside the monitor. Execution of the operations is guaranteed to be **mutually exclusive**, i.e. execution can only be by one process at a time. Thus, if one process reaches the execution of a monitor operation while another process is already using that monitor, it is suspended until the other process leaves the monitor, thus providing transparent synchronisation. Monitor constructs have been added to existing programming languages to form languages suitable for parallel programming, as in Concurrent Pascal [Brin77]; and formed the basis for new programming languages, such as Mesa [Lamp80].

An alternative model based on a shared-memory paradigm is the Linda model [Ahuj86]. The central concept in Linda is that of the **tuple space**, a shared memory space containing structured data items called tuples. Data can be placed in tuple space using a write operation, and read or copied out by any other process; indeed, processes can also be placed in tuple space, to be executed by the system at some time when a processor is free. Linda has been shown to be successful across a range of grain-sizes, on both shared-memory and distributed architectures, where it implements a virtual shared memory [Carr88]. Shared-memory models in general are particularly well suited to program structures in which the problem is broken down into sub-tasks, which can use different data to work on different parts of the overall computation, in other words result parallelism. Because Linda supports

process tuples, it also supports activity parallelism, with a master process generating data tuples and "slave" process tuples to consume them when processors become available.

Turning to programming models which are not based on shared resources, the archetypal message-passing model is known as the Communicating Sequential Processes model (CSP) [Hoar78] [Broo84]. This model relies on explicit send and receive operations used to pass data between two specific, named processes. At any one time, all data is owned by one process; this concept of a single owner is central. Synchronisation is achieved because messages cannot be received until they have been sent. The CSP model has been directly implemented as a programming language [Jaza80], and the Occam language is also based closely upon it [Jone85]. While most CSP-based languages allow only static allocation of processing tasks, a more recent language, Strand88, is a CSP-based system which supports dynamic work allocation [Catt89] [Fost90]. An alternative model based on asynchronous message passing is the Actor model [Agha86]. Asynchronous message-passing systems are particularly well-suited to applications in which pipelines of processes each pass on data to the next, in other words where there is structure parallelism.

While in CSP and its derivatives, communication between processes is asynchronous, the third type of model is based on synchronous message-passing. As in asynchronous message passing, all data has one owner; however, in synchronous systems, many data objects remain in the ownership of one process which acts as a caretaker, performing any necessary operations on them. Thus, when a process needs to perform an operation on some object, in this type of model the process must synchronise with the object's caretaker process (i.e., wait for it to return the result). This is the basis of the Remote Procedure Call (RPC), the remote procedure being the object's caretaker, and the call being synchronous. The Ada language's rendezvous mechanism is another example of this type of programming model [Mund86]. The two types of message-passing model, synchronous and asynchronous are philosophically similar, and it has been argued that they are directly equivalent in expressive power and performance [Laue79]. However, synchronous message-passing systems are designed more for client/server processing arrangements, and is therefore more suited to activity rather than structure parallelism.

As noted earlier, the fact that these parallel programming models have similarities to certain concurrent architectures does not mean they can only be implemented on such architectures. In the same way, the fact that each model is particularly suited to one of the three types of parallelism described above, certainly does not imply that programs based on one of these models cannot express any other type of parallelism. However, the fact that each of these types of programming model allows a certain type of flexibility that the others do not, implies that certain algorithms are easier to express in one model than another. Because algorithms which can only be expressed awkwardly tend to result in an inefficient implementation, in recent years there has been a move towards parallel programming models which are abstracted further away from architectural features.

One solution to this is to provide a model which consists of processes and a set of sequencing operations encompassing all three models above, so that whichever operations seem most appropriate can be chosen [Brow85c].

An alternative is to adopt an object-oriented programming approach and adapt it for parallelism [Toko87]. The object-oriented paradigm is convenient, since it involves encapsulating data and the operations which can be applied to it within self-contained units, which can therefore easily be distributed amongst processors. This distribution of objects can be done in two ways: either explicitly by the program (by inheriting the properties of parallel objects and synchronisation objects, as in Presto [Bers87]); or transparently by the system. In the latter case, the system must trap invocations of other objects if they are not on the same node. The two objects can then communicate either by message-passing, or by object migration - moving one of the objects so that they then reside on the same node. The management of objects placement can be done in various ways; for instance by tagging each object with a unique global identifier, as in the Emerald system [Jul88] or by maintaining a virtual shared object memory as the Amber system does [Chas89]. Systems based on object mobility have proved efficient, but the question of decision-making in such systems (when to migrate an object and when to leave it in place) is still a topic for further research.

While parallel object-oriented systems are typically aimed at distributed systems consisting of a number of von Neumann processors, one final control-flow based parallel programming model does not assume any one particular hardware

architecture. The Bulk Synchronous Parallelism (BSP) model is aimed at "providing guaranteed performance at near-optimal processor utilisation" [Vali90]. It does this by arranging for "parallel slackness" - in other words, ensuring that there are normally more virtual processes in existence than there are physical processors. In this way, all available processors can be kept busy processing the waiting tasks. This idea itself is not new; systems which are made efficient by judicious "grain-packing" of processes onto processors in order to balance execution and communication times have already been proposed [Boon88]. The main innovation of BSP is instead that its synchronisation facilities are not architecture-based. Instead, they are based on optionally controlling processing tasks by the use "supersteps" of a set period of time, after which the tasks are synchronised. This period can be controlled by software, at run-time if required, thus providing control over the effective grain size of the program (over a certain minimum necessitated by the hardware being used). Thus, BSP appears to provide a programming model which is abstracted away from traditional architectural models, while retaining the aspect of efficiency which is ever important in parallel processing.

Section 2.1 introduced parallel processing hardware, and this section has discussed a conceptual layer above the hardware, the abstract models involved in programming such hardware. The next step, therefore, is to move up another layer to examine the actual tools which can be used for parallel programming, based on these models.

2.3 Programming Tools

It has long been realised that parallel programming is sufficiently different from sequential programming that new, specialised tools would be needed for it, beyond simply providing parallel programming languages based on the models described in the previous section. In particular, the complexities introduced by concurrency give rise to specific programming needs unique to parallel programming [Wino79]. A number of software tools have been developed to aid the parallel programmer manage the complexity involved, but so far these have often failed to meet the needs of real application programmers [Panc90]. The following section examines

some of the tools which are available, and assesses what is needed. Section 2.3.2 then describes an alternative approach to the provision of tools to manage the complexity of parallel programming - namely, the application of techniques from the field of visual programming.

2.3.1 Tools for Parallel Programming

Perhaps the most common means of parallel programming is to use text-based HLLs; thus the most common tools used are compilers. There are three main approaches to providing compilers for parallel programming:

- To extend existing programming languages with parallel constructs, such as the Pisces system based on FORTRAN [Prat85];
- To provide "auto-parallelising" compilers which detect potential parallelism within a program and generate parallel code accordingly [Kuck81];
- To provide compilers for specifically parallel languages such as those mentioned in the previous section.

Many systems provide no tools for parallel programming other than the compiler, and standard debuggers not originally intended for debugging parallel program. However, other tools to be used in conjunction with parallel program compilers exist, particularly tools for inter-procedural analysis to discover data dependencies [Alle85]. There are also debuggers designed for parallel programming [Cheu90]. Some, such as Instant Replay [LeBl87], focus on ensuring reproducible behaviour; others, such as Parasight, on minimal intrusiveness [Aral88]. Although such tools ameliorate certain tasks, parallel programming still remains generally difficult.

Part of the problem may be that many of the existing tools each address only one aspect of parallel programming. Few systems provide an integrated computer-aided software engineering (CASE) environment encompassing the whole of the parallel software life-cycle (including design, implementation, debugging/testing and performance tuning).

However, the trend seems to be towards this ideal of a complete, integrated programming environment. The Pie system provides an architecture-independent programming environment based on a modular programming "metalanguage" on which tools for implementation and instrumentation (i.e. the collection of data for

debugging and performance tuning) are based [Sega85]. There are also systems based around existing languages; the Schedule system provides an integrated set of programming tools for a parallel FORTRAN [Dong87]. On a wider scale, a system called IPE provides an Integrated Programming Environment which is intended to be able to support all stages of the software development process for distributed parallel programming [Szaf92]. IPE is not based on one of the conventional parallel programming models, but instead uses a business organisation metaphor, in an attempt to make parallel programming more accessible. IPE is also aimed at large-grained computation, another trend in parallel programming tools as networks of workstations (which lend themselves only to the large-grain level of parallelism) become almost ubiquitous.

IPE makes use of graphics to specify and modify some aspects of the application. This, too, is becoming a trend in providing tools and even programming environments for parallel (and most other types of) programming. To understand why, it is necessary to examine the nature of visual programming.

2.3.2 Visual Programming

The term **visual programming** is used to refer to the design and development of software using visual images rather than text. The study of programming using images - in other words, computer graphics - is younger than that of parallel programming, because high-resolution bitmapped graphics technology has only relatively recently become so cheap that it is available to all. However, it has long been realised that graphical displays are inherently easier to understand than purely textual ones, for a number of reasons which include:

- Pictures contain more dimensions of expression - not only can they represent two or three-dimensional objects, but they can represent other physical properties such as size, shape, colour, direction and distance.
- Pictures resemble the real world - they are concrete rather than abstract. This means that the viewer can apply the same cognitive skills used to assess the world around us. These skills are used constantly, unlike language skills (and are learnt much earlier in life) and are therefore more familiar. This in turn makes the information transfer rate much faster [Raed85].

- Pictures are random-access rather than sequential-access - the eye can instantly move around a picture, and switch between viewing the overall picture and a detailed view easily.

It has been shown that, following from reasons such as those above, programming using visual abstractions of computer operations is easier than programming using textual HLLs [Powe83]. Moreover, it is the last of the three attributes above which make visual programming particularly well suited to the task of parallel programming.

From the principles described above, the field of visual programming has developed. Visual programming languages employ graphical representations of programs, in a self-consistent notation; a familiar example of a graphical representation of a program is the control flowchart, in which boxes represent operations and arrows between them represent the flow of control from one to the next. A notation can have formal properties (i.e., formal definitions of the syntax and semantics of combining aspects of the notation together) so that it is a precise definition of the behaviour of any program which can be expressed in it [Hare88]. To design or edit a program using a visual language involves a visual display and the **direct manipulation** [Schn83] [Card87] of easily recognisable shapes on it using appropriate input devices such as the mouse, trackerball etc. Entering and editing a program in a visual language is directly analogous to the same operations for a textual language, using a graphical editor rather than a text editor. Graphical editors which allow the manipulation of graphical images (essentially moving pixels on a bit-mapped display) are not significantly more complex than text editors.

Visual programming is beginning to have a major impact on programming methods. Among its advantages is that complete programming environments can be built around a visual abstraction, including integrated toolsets covering several stages of the software development cycle [Ambl89]. But not only have visual programming techniques made programming easier, in the field of parallel programming they have begun to make things possible which were previously infeasible, due to the difficulty of managing the complexities of parallelism.

Visually-based tools for parallel programming have begun to appear. These tools can be divided into three main types [Roma89]:

- visual programming environments (in which algorithms are specified);
- program visualisation and animation tools (graphical representations of algorithms and their run-time behaviour); and
- data visualisation and animation tools (graphical representations of data structures and their run-time behaviour).

Taking each of these in turn, visual programming environments for sequential languages have been under development for some years. One of the earliest was the Cornell Program Synthesiser [Teit81] which, while not completely graphical, introduced the concept of syntax-directed entry of application code; the programmer's input being restricted to the syntactically correct possibilities. This has become a common feature of software tools which support visual programming. Through the 1980s, many visual programming environment systems have become available [Chan87]. Those programming environments designed for parallel programming are only a fraction of the total, but there have been a number of successful systems among them. Some have provided visual tools designed to produce code in an existing textual programming language, such as graphical Occam [Mour89], which is based on control flow graphs which illustrate Occam's constructs for parallelism and its communication channels. Others have developed new visual abstractions for parallel programming, at a variety of levels, for example Poker, which uses graphs to describe the architectural characteristics of distributed systems [Snyd82]. At a more architecture-independent level, CODE provides an Computation Oriented Display Environment in which graphs represent the data and scheduling dependencies between processes [Brow89]. Other systems have adopted existing architecture-independent models, such as the object-oriented model [Roge88]. At a higher level still, PegaSys uses graphics which represent statements in a formal logic to represent the specification and design of parallel programs [Mori85] [Mori86]. Another system, Faust provides a spectrum of tools from high-level project management tools, to low-level event displaying tool [Guar89].

It was noted above that visual programming tools can be classified as those for algorithm visualisation and those for data visualisation, and many of the tools provided by visual parallel programming environments such as those mentioned above fall into these categories. In addition, there are a number of such tools which do not form part of a complete visual programming environment.

The purpose of algorithm visualisation tools which are not integrated into a programming environment per se is to generate graphical representations (or "visualisations" of parallel programs developed in another way. There are many possibilities for algorithm visualisation [Brow85]. For instance, recent developments of the Pie system provide visual depictions of algorithmic behaviour designed for performance debugging [Sega89]; the Voyeur system provides hierarchical graphical views of Poker programs [Soch89]; views of parallel programs can also be generated and viewed using an entity-relationship based query language [Schw86]. The motivation for these tools is that if programmers have access to a rich variety of ways of viewing the parallel program, they will be better able to understand the complexities of the algorithm and hence of the program's behaviour.

To represent the program's behaviour at run-time is the aim of algorithm animation tools. Such tools are based on some other graphical representation of the algorithm and add to it graphical cues (such as colour changes) which represent the activity of the program, such as the position of threads of control [Brow85b]. Of more interest are data visualisation tools, since the contents of data structures are generally much more complex than the flow of control, and therefore the programmer benefits more from visualisations of data. The VIPS system provides both algorithm and data visualisation tools for Ada programs [Isod87]. Since data structures are defined by the programmer, the programmer also specifies the most appropriate way to display each variable required, using a Figure Description Language. The visualisations can be chosen from a predefined set; or in Incense, a data visualisation tool for the Mesa language, can be defined by the programmer [Meye83].

The theme common to all systems which provide data visualisation tools is that understanding the flow of data through a program is central to the understanding of the program's behaviour. Indeed, the flow of data has been used as the central visual concept in many sequential visual programming languages. In parallel visual languages, the relationships between processes and in particular their synchronisation has tended to be the focus. However, the dataflow paradigm does allow the visual specification of synchronisation, as a few systems have proved. They are described in the next section, which examines dataflow in more detail.

2.4 Dataflow

The basic concept of the dataflow paradigm was introduced in Chapter 1. The idea of describing programs in terms of the flow of data through them has been around almost as long as that of describing them in terms of flow of control. One advantage of the dataflow approach is that data items usually represent real-world entities, and so to describe the program in terms of these (rather than in terms of an artificially devised flow of control) makes the program description closer to the real world, and hence easier to understand. For this reason, "data flow diagrams" have long been used for systems analysis, to describe the overall design of a program though not the implementation. It is also because of this closeness to the real world that dataflow is popular in the field of visual programming, since relating programs to the real world makes them more understandable - the chief goal of visual programming. However, most of the interest motivating research into dataflow for its own sake has been due to the implicit parallelism found in dataflow-based descriptions of programs. The following sections describe the field of dataflow research, which is in itself a complex area with a number of distinct branches.

2.4.1 Dataflow for Parallelism

As explained in Chapter 1, dataflow programs consist of discrete operations, which have data inputs and outputs. Dataflow programs are almost always considered as digraphs - directed graphs in which the operations are the nodes, and data items flow in a set direction along the arcs of the graph. These arcs describe *all* of the data dependencies which exist between operations; the only data to which operations have access is that of their inputs, there being no global memory. From this, it is implied that any two operations which do not have an explicit dependency have the potential to be executed concurrently, since they do not share any resources for which there could be contention. Indeed, even if the operations are not ultimately executed in parallel, the lack of any shared information between them means that there can be no **side-effects** (one software module interfering with another's data, causing results not obvious from looking at either module). This freedom from side-effects is a very useful property even when developing sequential software [Stev82]. In a parallel environment, where debugging

information is often harder to collect and interpret, this becomes even more important.

It has been shown that this implicit parallelism is straightforward to extract from dataflow descriptions of programs, for use in executing the operations of the program in parallel [Trel77]. Thus, on architectures on which concurrent execution is possible, it is often useful to use a dataflow rather than control-flow description of a program; in other words, to use dataflow not just as a design tool, but for the implementation as well. Indeed, the time at which a particular interest began to be taken in dataflow as an implementation medium in its own right, was the early 1970s when a research project at MIT undertook to implement an early parallel architecture based on the dataflow paradigm.

Research into dataflow machine architectures is described in the following section, but from the early 1970s onwards, there was increasing interest in the dataflow paradigm at all levels of software development - at the design level and at the programming language level (described in section 2.4.3), as well as at the machine architecture level. Much dataflow design work arose from the dataflow system originally devised to show how the proposed MIT dataflow architecture would be programmed [Denn74]. This system consisted of a design notation based on dataflow graphs, not dissimilar to the example dataflow graph in section 1.4. In this system (and its successors), data is introduced into the program, and flows through the graph. The dataflow operations (operations in this case meaning machine instructions such as addition, multiplication etc.) are executed when all of their inputs are present. This is usually known as **firing** - operations are fired when they have data available on all inputs (and when processing hardware is available to execute the operation). Each operation has two states: **enabled**, which means that data are available on all inputs, so the operation can fire as soon as the appropriate hardware is available; and **not enabled**, when data are not available on all inputs. The system of waiting until all inputs are available before becoming enabled is known as the **strict enabling rule** and most dataflow systems have followed this, for simplicity of implementation.

The type of dataflow system described above is known as **data-driven** since the fact that data are available drives the events. This is the scheme used in most dataflow systems; however, an alternative exists, in **demand-driven** dataflow.

Here it is demands for the data needed which drive the events (enabling operations which would produce the data required, which may in turn enable further operations to produce the data they require, and so on). The obvious analogy is that data-driven dataflow corresponds to the imperative language model, while demand-driven dataflow corresponds to the functional language model. In data-driven dataflow, data is "pumped" through the graph, while in demand-driven systems it is "sucked" through. Some systems of dataflow allow both interpretations [Shar82], but data-driven dataflow is the more common, particularly in the research into dataflow architectures. This research is now discussed in more detail.

2.4.2 Architectures

Dataflow architectures are designed to provide a parallel alternative to the inherently serial von Neumann model of processor architecture. Such architectures exploit the potential parallelism of dataflow programs by executing operations which do not have immediate dependencies concurrently on their multiple processing units. These processing units are "fed" from a queue of enabled operations, and because the concurrent operations do not depend on each other for sequencing, the processors can execute them asynchronously. Items of data are passed between modules of the architecture as **data tokens** containing the data value and the destination operation of that data item; operations are passed as **operation packets** comprising the opcode, operands and destinations of their results. This packet communication system makes dataflow machines particularly suited to distributed organisation, and makes dataflow architectures easily scaleable.

There are two main types of dataflow architecture. In the MIT machines and others, only one data token is allowed to exist on each arc of the dataflow graph at a time [Denn80] [Denn88]. This form of dataflow avoids problems of sequencing successive data tokens on the same arc, and has been used for its deterministic properties [Rumb77]; it is known as **static dataflow**. This type of system involves **feedback** since if an operation's output arc remains full, that operation cannot fire again if it would produce another data token for that arc. This information about the full/empty state of each arc must be fed back by control tokens.

In **dynamic dataflow**, on the other hand, multiple data tokens are allowed to exist on the same arc. Dynamic dataflow machines, such as those developed at

Manchester [Gurd80] and by a second team now at MIT [Arvi90] use a **tagged-token** system in which each data token carries a unique tag to identify its sequence on an arc. This removes the need for feedback and allows **loop unravelling** - successive iterations of loops can sometimes be executed in parallel rather than in sequence, the results being restored to the correct order using the tags. Thus, dynamic dataflow architectures provide greater potential for the exploitation of parallelism in a program [Gurd86].

However, this ability to exploit large amounts of fine-grained parallelism is not the unqualified advantage it might seem; it is often the case that there is more potential parallelism in dataflow programs than the architecture can handle, leading to very long queues of enabled operations and high memory usage of data tokens produced. Thus, a need has been identified to *reduce* the parallelism in programs for dynamic dataflow architectures [Bic87]. This and other problems, such as the cost of large packet-switched networks needed when many processing units are implemented, are ongoing topics of dataflow architecture research.

Various alternatives to the classic static or dynamic dataflow architectures have also been proposed. For instance, it has been suggested that dual architectures would be possible, with instructions activated either by control tokens or data tokens, according to context [Hopk79]. Most others have been hybrid architectures, based on cheap von Neuman processing elements, but with dataflow synchronisation of the operations [Trel82] [Gaud85] [Bueh87]. Such architectures generally use a slightly coarser-grained definition of operations than the instruction-level dataflow architectures; hardware support for dataflow is used because of the ease of extraction of parallelism, and freedom from side-effects. Similar architectures have implemented dataflow "macros" which allow groups of instructions to be scheduled as a single dataflow unit or **actor** [Evri90]. These architectures still provide fine-grained parallelism, but there seems to be a significant trend towards coarser-grained parallelism.

This is not to say that fine-grained dataflow architectures have no future; work on both static and dynamic architectures continues [Gurd87], and there is hope that the main technical problems with such architectures may soon be overcome [Nikh89]. However, any trend towards larger grain sizes is relevant here because it shifts the focus towards a higher level of abstraction; in this case, away from

hardware and towards software. The software uses of dataflow, therefore, are described in the following sections.

2.4.3 Low-level dataflow languages

A number of textual dataflow languages, somewhat akin to the HLLs used to program conventional architectures, have been developed. This section provides a only a brief overview, in order to provide context for the later discussion of higher-level dataflow systems; the details of these languages are not directly relevant.

Many of these textual dataflow languages follow the pattern of imperative HLLs, but with important differences, notably that of **single assignment**. Languages such as ID [Arvi77], Lucid [Wadg85] and SISAL [Cann90] do not have variables, which can be modified at any time, but simply **values**, which can only be assigned once. Values model the arcs of dataflow graphs; they are set by the operation which produces the value, and read by the operation which consumes it. These languages can be compiled into code for dataflow architectures, which then use the assignment of values to sequence the operations.

Programming languages based on the dataflow paradigm have been called applicative languages [Acke82]. A number of hybrid languages have also been proposed, combining dataflow and control-flow features; for instance applicative languages which allow variables rather than values [Ruig90], or languages based on path expressions [Oldh84] or CSP [Bond89] [Gaud89]. The variety of programming languages used is almost as great as the variety of broadly dataflow-based architectures.

Not all dataflow-based languages have been developed specifically to program dataflow architectures; the dataflow paradigm has also been used simply for its advantages as a programming model and software engineering tool, namely the ease with which parallelism can be identified and freedom from side-effects [Gajs82]. An example is Mentat, an object-oriented programming system based on a dataflow model [Grim87]. Another is the Tyger model, a hybrid between dataflow and control flow which employs alternating "stripes" of dataflow and control-flow execution, with shared values passed between the two [Stok90]. A further benefit of the dataflow model which these systems and others exploit is its architecture-independence, making it easy to port dataflow programs between

different types of parallel architecture [Ski190].

As mentioned earlier, the details of these text-based dataflow languages are not important; what is significant is their place within the context of dataflow programming. These applicative languages and others form the implementation language for dataflow programs, at a higher level than the machine instructions needed to manipulate the target architecture, but at a lower level than the overall program design. Having briefly outlined the work which has been done in the first two of these areas, we can finally turn to the last, the area of dataflow program design.

2.4.4 Dataflow design languages

The obvious vehicle to use for designing dataflow programs is the dataflow graph. The advantages of using graphs for program design are the same as those of visual programming: graphical representations of algorithms are simple to understand. In the case of dataflow, this is particularly evident in the visual representation of implicit parallelism, since potential parallelism can be seen wherever two operations do not have direct dependencies.

A number of notations for dataflow graphs have been proposed; the original motivation for these was to provide visual design languages for programs to be implemented on dataflow architectures [Kosi73] [Denn74]. For this reason, a number of variations arose which reflected design decisions within the proposed architecture, such as specialised arcs for control tokens [Woo83]. However, the basic features such as digraphs, arcs representing the paths for data, nodes representing operations, and so on, remain common [Davi82]. Another feature which many notations include is the ability to include cycles in the graph; although allowing data tokens to loop back tends to introduce nondeterminacy, it allows operations to be given information about the past; this can only be done with loops since operations themselves have no persistent state between firings. This ability to express "history-sensitive computations" by allowing graph cycles has proved useful.

Besides these basic concepts, the graph notations include specialised features corresponding to programming constructs in lower-level languages, allowing selection, iteration and so on. Examples of specialised features found in some of

these graphical notations are:

- replicators, which copy one input data token to two output arcs;
- unions (merges), which pass on a token from one of a choice of input arcs to one output arc;
- selectors (switches), which pass on one of several possible input tokens according to the value of a control input;
- case nodes, which pass on their input token if it has a certain value;
- loop nodes, which pass on their input data token while a control input remains true.

These operations are equivalent in grain size to the fine-grained operations implemented on dataflow architectures. However, similar graphs continued to be used when the usefulness of applying dataflow techniques to non-dataflow architectures began to be recognised - in other words, in a less fine-grained environment [Shar85]. In such cases, fewer specialised features of the notation need be provided, since the programmer generally implements the operations themselves, rather than having to rely on what operations are available in the hardware. Operations provided by the programmer within a dataflow graph are generally called dataflow **actors** (with reference to the actor model, described in section 2.2.3).

It was primarily the move towards distributed computing of the 1980s which motivated research into dataflow at a larger grain size. In the LGDF (Large Grain Data Flow) language [Babb82], [DiNu88], dataflow graphs can be designed in which the operations are whole modules of the eventual application program. Programs designed in LGDF are intended for parallel execution across a network of workstations, hence the need for a large grain size because of the relatively high cost of communications. While still essentially a design language, LGDF2 [DiNu89] moves even closer to doubling as a high-level implementation language; TDFL (Task-level Data Flow Language) [Suhl90] is another system with the same objective. In these systems, while the program modules are still coded in some textual HLL, the overall design (i.e. the dataflow side of the program) is expressed in the high-level language. Because it is straightforward to detect parallelism in a dataflow program, it should be easy for software to take a machine-readable form of the dataflow graph, and combine it with the module code (the dataflow actors) to

form an executable parallel program. This is a powerful idea because it allows programmers to generate parallel programs without having to write parallel code, thus avoiding the pitfalls of parallel textual HLLs. Because of this, the programmer's own code remains portable - only the tool which generates the parallel code need be modified for different platforms. The task of generating the parallel code is shifted from the programmer to the system, with the aid of a visual CASE tool for parallel programming.

One system which already provides such a tool is Paralex [Baba91]. Paralex programs use a very simple (acyclic) dataflow language for the program design, which can be edited using a graphical editor. The actors are coded in textual HLLs using traditional text editors. The system then compiles this into a set of program modules which can be distributed across a group of interconnected workstations. Although the primary goal of Paralex is fault tolerant computing through replication of the actors, it provides an effective way of parallelising very computationally intensive programs.

One important aspect Paralex has in common with design languages such as LGDF is that it is intended for a very large grain size - that appropriate to networked workstations. This is reflected in the design of their dataflow notations, which have no specialised operations; the languages provide *only* very high-level support for dataflow execution, and lower-level operations such as input/output, replicating data items, and merging streams of data items must be done by the programmer within a module.

To summarise, applicative languages and others cater for fine-grained dataflow (on specialised architectures); there are systems based on textual HLLs for dataflow execution at a medium-grained level (on conventional multiprocessor architectures); and there are visually-oriented design tools for very large grain dataflow (on distributed systems). However, as yet no visually-oriented dataflow design system has been proposed which caters specifically for medium-grained multiprocessors of the kind described earlier in this chapter.

2.5 Summary

This chapter began by discussing multiprocessor technology, and in particular

shared and distributed-memory multiprocessors. These were described in the context of the other forms of parallel processing equipment, and in terms of the software grain size which they support. Then, the methods by which they can be programmed were discussed, in terms of the programming models upon which parallel programming languages are based. Some of the tools which can be used to aid parallel programming were also described, with particular emphasis on visual programming. As noted in section 2.3.2, visual programming is particularly well suited for parallel programming, because visual abstractions are easier to understand and work with than textual abstractions - a fact which is particularly important given the high degree of complexity of concurrent programs of non-trivial size. The trend towards integrated programming environments was noted.

The dataflow paradigm was then described in detail. Dataflow is inherently a visual model, and it was mentioned that for this reason, a number of sequential visual programming languages have been based on dataflow concept. Although dataflow graphs were originally intended for the design of fine-grained dataflow programs, they have also been used as a visual programming language at the large-grained level.

This thesis seeks to fill the gap in current dataflow programming research between the fine-grained and large-grained levels of parallelism. The next chapter introduces a new form of dataflow graph designed with efficient execution at the medium-grained level in mind, and intended for use as a visual language; and indeed, as the basis of an integrated CASE environment, supporting the development of parallel software from the design stage, through implementation and debugging to performance tuning.

For reasons of scope, this thesis examines the effectiveness of the new notation only in terms of efficiency of execution, and does not include an evaluation of its use as a visual design and implementation language. The possibilities for such an evaluation are discussed in Chapter 6. Moreover, this research is limited to the consideration of dataflow programs on conventional MIMD multiprocessors. This chapter has sought to show that there is a need for a visual design and programming language for medium-grained multiprocessors; the remainder of the thesis seeks to demonstrate that dataflow is a feasible candidate for such a language.

The MeDaL Notation

This chapter introduces MeDaL (an acronym for *Medium-grained Dataflow Language*). The MeDaL notation has been developed as a visual way of specifying parallel programs, and in some respects resembles the various dataflow notations described in the previous chapter. However, MeDaL is unique in being designed specifically for the medium-grained (inter-task) parallelism commonly used for parallel programming on the medium-grained MIMD multiprocessors described in Chapter 2. This chapter describes MeDaL and the decisions taken in its design, in relation to many of the issues of parallel programming discussed in Chapter 2; Chapter 4 goes on to describe how MeDaL can be implemented to allow the efficient implementation of parallel programs.

MeDaL, however, is not designed purely to be a visual specification of parallel programs. Rather, it is designed to fulfill some of the rôles of a software development methodology, supporting the development of parallel software from the design phase, through implementation and debugging and testing, to performance analysis and tuning. This thesis concentrates on the implementation phase, though tools based on MeDaL for supporting the other phases are described in Chapter 6. However, these wider aims of MeDaL should be borne in mind and will be referred to in the discussion of some of the design decisions.

This chapter is structured as follows: firstly, the main concepts of the MeDaL notation are described, to provide an understanding of the MeDaL approach to describing parallel programs. Then the semantics and syntax of the notation are described in detail, drawing on these concepts. Having presented MeDaL, a number of the key design decisions that were made during its evolution are discussed, and a number of the alternative ideas which were considered but rejected are described. It is envisaged that any full implementation of MeDaL would include a library of useful functions which could be incorporated into MeDaL programs, and a minimal suggested library is also described. An example MeDaL program is then presented, and its salient features discussed. The implementation and performance of this

program is described in Chapter 5.

The final section of this chapter gives a reprise of the main features of MeDaL, and shows how MeDaL achieves its aims as a flexible parallel program specification language.

3.1 Concepts

This section describes the specific aspects of dataflow used in the MeDaL notation. MeDaL is a traditional data-driven dataflow notation in the tradition of those described in section 2.5. Programs are specified in the form of a MeDaL graph, consisting of nodes (called dataflow **actors**) and lines connecting them (called **datapaths**). Data can only flow through datapaths in one, specified direction, but loops in the graph are allowed, so MeDaL graphs are cyclic digraphs. MeDaL follows the dynamic dataflow model in that more than one item of data is allowed to exist on a datapath at the same time.

With specific exceptions, MeDaL actors obey the strict enabling rule; they fire only when data is available on all input paths (and since datapaths are directed, each path is an output path for one actor and an input path for one actor). Once data has initially entered the graph, actors become enabled when at least one item of data is available on each of their input paths, and fire repeatedly (normally consuming one item from each of these paths), for as long as this condition continues to be met. When this condition is no longer true, the actor becomes disabled once it finishes processing its last complete set of inputs, but may become enabled again later. Execution of the program terminates when no actor is in an enabled state (or until the program is halted artificially). Although this firing rule is the fundamental basis of the execution of MeDaL programs, there are a number of variations to it, provided for greater flexibility of programming, and which form the basis of much of the MeDaL notation and semantics.

MeDaL is designed for medium-grained parallel programming, and therefore only describes algorithms at the medium-grained level, in other words, at the sub-program level. MeDaL actors are tasks, such as subroutines within an application program; the tasks themselves are not described by the notation. Like the large-grained dataflow systems described in Chapter 2, the graphical representation of an

actor corresponds to a section of code written in some other language; in the case of MeDaL, it corresponds to a single function or procedure. It is envisaged that actor code will normally be written in some textual imperative language such as a C or C++ function, a Pascal procedure or a FORTRAN subroutine. The interface to these code sections, in an abstract form, is included in the specification of MeDaL. The code section which forms an actor in MeDaL is known as a **method** (following object-oriented terminology) and an actor which contains a method (all actors contain zero or one methods) is known as a method actor.

MeDaL consists of six kinds of actor, which can be divided into two types: general-purpose and special-purpose. The function of general-purpose actors is not specified by the notation, and hence all such actors contain a method. The special-purpose actors are, as the name suggests, provided for a specific purpose and does not necessarily contain method code supplied by the programmer; but in some cases, a method can be supplied if the programmer wishes, and these cases are described below.

The components which make up the MeDaL notation consist purely of closed shapes and lines which connect them, but textual annotations are recommended to aid the programmer in relating the MeDaL graph level of the program to the actor-method level. In particular, all method actors should be labelled with the name of the method (as used in its declaration in the underlying code). All of the illustrations of MeDaL features and the examples in this thesis include annotations, but it should be borne in mind that they do not have any syntactic effect on the notation itself.

MeDaL programs can be given a modular structure; to follow the acting metaphor, these modules (consisting of a collection of actors and datapaths) are called **companies**. A program comprises one or more companies. These companies form a tree structure, with the first company at the root of the tree, and any company can contain any other company (except the root company), including itself, thus allowing recursion. Companies serve as the visual equivalent of a macro. When executed, a MeDaL graph is created from the root company, and subsequently whenever data flowing along a datapath encounters a company, the graph used for execution is rewritten to include the graph contained in that company. This hierarchical structure encourages the top-down structured design of

programs using the notation.

With these points in mind, it is now possible to describe the notation itself.

3.2 MeDaL Semantics and Syntax

As outlined above, MeDaL programs consist of several components:

- Companies, which contain:
 - Datapaths
 - Actors, which may contain:
 - Methods

This section describes the semantics and visual syntax of datapaths and then actors. The actor methods interface with datapaths in various ways, and these interfaces are described in the appropriate sections. Finally, the syntax and semantics of companies are described.

3.2.1 Datapaths

Semantics

A datapath can be thought of as a unidirectional buffer between a producer and a consumer. Data items are queued in FIFO order and cannot overtake each other. The data items must all be of the same, specified data type, but this data type need not be an atomic type; structured data types such as records and arrays can be passed as single items, for instance.

There are two types of datapath, called E-type and F-type paths (a mnemonic for empty and full paths). E-type paths are used in most circumstances, and implement a classic FIFO queue in which, when a consumer reads the head of the queue, that data item is removed from the queue. F-type paths are similar, except that *if* the tail of the queue is empty, the item at the head is *not* removed, only copied; it remains in the queue for as long as it is the only item in that queue, and may be consumed any number of times. This implements a form of "persistent" memory, in the sense that the item is stored persistently relative to the number of firings of the consumer actor. This is important because actors themselves are not intended to retain any state (such as persistent memory) between firings.

Syntax

A datapath is represented as a line between the producer actor and the consumer actor. The direction of flow is indicated by an arrowhead at the consumer end. The arrowhead is coloured white in the case of an E-type path, black in the F-type case (Figure 3.2a). In addition, the ends of the datapath line can only touch the "southern" edge of its producer actor, and the "northern" edge of its consumer. Thus, the overall flow of data is north to south, but datapaths are not constrained to being straight north-south lines, and effectively data may flow in any compass direction so long as the datapath connects an output to an input.



Figure 3.2a: Datapath syntax

Method interface

Datapaths can only be manipulated by the method code in the producer actor. The actor can `send` a data item on a particular output path, adding it to the path's queue, which may cause the consumer actor to fire immediately. The producer can also `flush` a particular output path, destroying all data in the queue, ensuring that the next item to be sent on that path will immediately be the head of the queue. This is particularly useful in conjunction with F-type paths.

There are two further operations on datapaths. `Send-sticky` acts like `send`, placing a data item on an output path; however, being "sticky" means that it cannot be delivered to the consumer - even if it reaches the head of the queue - until re-sent using `send`. In conjunction with the method interface with the actor (see below), this provides a powerful form of persistent memory between firings (though unlike the persistence of the last item in an F-type path, the stickiness mechanism requires the intervention of a method actor). This mechanism is described fully in section 3.4. Finally, a predicate function `is_sticky` is provided which for a given output path, returns the boolean *true* if the tail item of that path is "sticky", and *false* if not.

For the purposes of the notation, datapaths are assumed to be infinite in storage capacity; in other words, `send` operations always succeed, and never "block" the

producer. If the system running a MeDaL program runs out of memory it is assumed to be fatal to the MeDaL program (as it would be to any other application).

3.2.2 Actors

General Semantics

Actors have 0 or more inputs and 0 or more outputs, each input or output being connected to a datapath in accordance with datapath semantics. Like datapaths, actors' inputs and outputs are typed (only carry items of one data type), and it follows that the type of each input or output must match the type of the datapath to which it is connected.

When data arrives on a datapath for a particular input, this may cause the recipient actor to become **enabled**; the definition of enabled varies between the six different kinds of actor presented below. At some point in time after the actor becomes enabled, it begins execution or **fires**. Firing denotes the start of execution of some code associated with the actor: either a method supplied by the programmer, or in the case of the special-purpose actors, code provided by the system to carry out a predefined action in the absence of a method.

All actors are assumed to terminate. In most cases, an actor can then be fired again (after termination) if data is again available on all input paths.

General Method Interface

An actor's only interaction with its input paths occurs at the point when it fires. At this time, it effectively performs a `read-input` operation on each of its input paths, taking the data item at the head of each datapath's queue (which may or may not remove it from the queue - see the section on datapath semantics above). Also at firing, the actor effectively performs a `read-output` operation on each of its output queues, which returns an empty data item of the appropriate type if `send-sticky` has not been used on that output, or if `send-sticky` was used, returns the "sticky" data item sent. All inputs and outputs read in this way are then passed to the method (or internal) code which begins executing. This procedure is automatic, and only the results are visible to the programmer.

Although these read operations occur only once, when the actor fires, actors are allowed to place data on output paths using `send` etc. at any time during execution,

in order to allow any actors which may depend on them for data to fire as early as possible.

This programming interface applies to all six types of actor provided by the MeDaL notation, and the particular semantics and syntax of each of these are now presented. The six types are: general-purpose actors, deep actors (a subset of general purpose actors), source actors, sink actors, merge actors and replicators.

General-purpose Actors: Semantics

The general-purpose actor has one or more inputs and one or more outputs. There is no default action, so general-purpose actors always contain a method supplied by the programmer.

The standard general-purpose actor follows the general semantics above. However, there is also a special type of general-purpose method actor called the **deep actor** (the reason for the name will be revealed in section 3.5). Multiple instantiations of the same deep actor can exist at the same time; for instance, if there are three or more items of data in each of its input paths, a deep actor can fire three times immediately, rather than waiting until the method has terminated before firing again. Each instantiation consumes the head of each input queue, and may then execute concurrently with the others; however, the act of firing itself (and the reading of the heads of input queues) is not concurrent. Moreover, making a method actor a deep actor does not guarantee concurrency, only allows it if sufficient processors happen to be available. It should be emphasised that only one copy of each general-purpose actor can be executing at a given time unless it is explicitly a deep actor; in this respect, MeDaL encompasses both the static and dynamic models of dataflow.

General-purpose Actors: Syntax

A general-purpose actor is represented in the MeDaL notation as a round-ended box as shown in figure 3.2b. The input/s arrive on the north edge and the output/s leave from the south edge. The box contains a label indicating what task it performs.

Deep actors are denoted by having a shaded border round the outer edge.

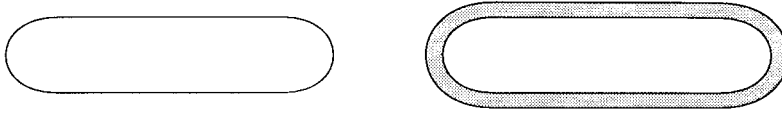


Figure 3.2b: General-purpose actors

Source Actor: Semantics

The source actor is the only type of actor with no inputs. A source actor fires only once, when the company which contains it is expanded (at the start of the program's execution in the case of the root company) - see section 3.1. If no method is supplied by the programmer, the default action is to place the boolean value *true* on its output path and then terminate. It can have only one output path. Since all other actors cannot fire until data arrives on their input path/s, it follows that all useful MeDaL programs must contain at least one source actor.

Source Actor: Syntax

The source actor is drawn as a polygon made from a square above a right-angled triangle with the right-angle pointing south. The output leaves from the point of the triangle. The polygon box may contain a label, especially appropriate if the actor has been given a method.

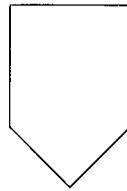


Figure 3.2c: Source actor

Sink Actor: Semantics

The sink actor is the only type of actor with no outputs. It has one input. The default action after firing is to do nothing, so that the data arriving on the input path (and being consumed when the actor fires) is lost. However, a method can be provided, for instance to release memory or other resources used by the data item.

Sink Actor: Syntax

The sink actor is represented by a polygon made of a square below a right-angled triangle pointing north. The input arrives at the north point of this triangle. The box may contain a label.

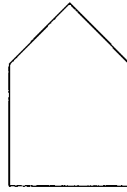


Figure 3.2d: Sink actor

Merge Actor: Semantics

The merge actor is the only type of actor which is not subject to the normal firing rule for actors. It has two or more input paths, and one output path, all of which must be of the same data type. The actor fires when a data item arrives on *any one* input path, consuming that data item only. If two items arrive simultaneously on two paths the actor will fire twice, consuming the items one at a time; but the order in which they are taken is not defined. Thus the merge actor simply copies the input/s to the output. It cannot contain a programmer-supplied method, for reasons explained in section 3.3.2.

Merge Actor: syntax

The merge actor is a closed semicircle, with the straight edge on the north side. The inputs arrive on this north edge and the output leaves from the southmost point of the semicircle. (In Figure 3.2e a merge actor with two input paths and its one output path is illustrated).

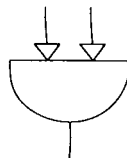


Figure 3.2e: Merge actor

Replicator: Semantics

The replicator is essentially a specialised form of the method actor. It has one input path and two or more output paths, all of which must be of the same data type. The replicator's function is to copy the input to all of the outputs. It cannot contain a method (again, the reason for this is discussed in section 3.3).

Replicator: Syntax

The replicator actor is a right-angled triangle pointing north, as illustrated in Figure 3.2f. The input path arrives at the north point, and the outputs leave from the south edge. (In Figure 3.2f it is illustrated with its input path and two output paths.)

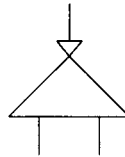


Figure 3.2f: Replicator

3.2.3 Companies

The final visual aspect of the MeDaL notation is that of the company. As outlined in section 3.1, a company is essentially a representation of a hierarchical module of a MeDaL program. Each company can be viewed in two ways: as a component of another company, or as a MeDaL graph in its own right. Naturally, the two different logical views of a company means that two graphical representations are needed.

Viewed as a **component**, a company simply replaces part of a graph. A company can appear anywhere in a MeDaL graph, in place of a number of actors and datapaths; it is a visual macro, analogous to textual macros in textual programming languages. Graphically, it is represented as a rectangular box (which may contain a label indicating the name of the company). To be useful, at least one datapath will enter any company. Input paths enter via the north edge, but since the company is not itself a destination to which data can be delivered, the input paths do not have arrowheads here. The company may also have outputs, which leave from the south edge. Figure 3.2g illustrates a company with two inputs and two outputs; in a real MeDaL program, it is likely that the inputs and outputs would be

annotated to permit association with the exploded (graph) view.

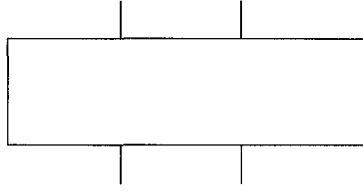


Figure 3.2g: *Company (component view)*

Viewed as a **graph**, the company is represented as a larger rectangular box, containing other components of the MeDaL notation; indeed, all MeDaL components appear within a company. It should also contain a textual label indicating its name (the name of the program itself, in the case of the root company). Companies which are components of another company will have at least one input path, which enters from the north side of the box, and if there are output paths, they leave to the south side of the box. An example of this notational device can be seen in the matrix multiplication example in section 3.4.2.

The numbers of inputs and outputs to a company, and the left-to-right ordering must remain the same for both of its two views: for example, the leftmost datapath seen entering a company when viewed as a component, is the leftmost path to emerge from the north side of the box when the company is viewed as a graph.

It should be noted that at run-time, the graph contained in a company is added to the graph of the main program the first time that a data item is sent to a datapath which crosses into that company (it is also at this point when any source actors contained in the company are fired). The fact that companies are used for dynamic expansion of the graph at run-time means that recursion is possible. Thus, companies play both a structural rôle, in terms of organising groups of MeDaL components into a hierarchical structure, and a dynamic rôle, in providing groups of components to be executed at run-time.

3.3 Design Decisions

This section examines some of the main design decisions which were made during the process of designing the MeDaL notation. After briefly justifying the visual syntax, a number of details of the semantics are examined: firstly the semantics of

the various types of actors, then those of datapaths. The least clear-cut design decision involved was the form which persistent memory should take, and section 3.3.4 describes a number of the alternative mechanisms for this which were considered, along with possible visual representations which could have formed part of the notation. It is likely that any future research based on MeDaL would wish to re-evaluate the design criteria explained in this section, so as full an account is given as possible.

3.3.1 General Visual Syntax

The shapes of the MeDaL actors were chosen primarily to be as visually distinctive as possible. It is the shapes rather than the sizes which distinguish actors, but it is assumed that the actors will be drawn with the relative proportions illustrated in section 3.2.

The merge and replicator actors are the shapes proposed by Sharp [Shar82]. The round-ended boxes of the general-purpose actors correspond to what are usually shown as circles in more traditional notations such as Sharp's, but it was found that an elongated shape was convenient for longer, more descriptive textual labels - necessary in MeDaL to relate the visual level to the hidden method level.

The source and sink actors are novel to the MeDaL notation, and the shapes were chosen simply to be distinctive, emphasising the special semantics of these actors.

The use of arrowheads on the datapaths is not, strictly speaking, necessary in MeDaL since the notation specifies that where a datapath connects to a south edge or point, it is an output, and when a north edge or point, it is an input. However, it was found that the use of arrowheads reinforced this directional convention, making the graphs easier to read. Additionally, some means of visually distinguishing E-type and F-type datapaths was needed. One possibility would have been to use dashed or dotted lines, but to incorporate this in the colouring of the arrowheads was preferred because this difference in datapaths, after all, only has an effect on the consumer, which is where the arrowheads are located.

3.3.2 Actor Semantics

This section briefly justifies the provision of the six different types of MeDaL actor. Some notations have chosen to have only one type of actor, while others have provided a wide variety; this decision is essentially a trade-off between providing useful functions (which, being part of the system, can be implemented for optimum efficiency), against providing generality and a simple programming model. MeDaL attempts to compromise between these goals.

For instance, the replicator and the merge actor are provided as useful functions. They have proved useful in designing applications using MeDaL; it is regularly necessary for one data stream to be copied to several different actors, and for several streams to be merged to lead to one destination (for instance when several sections of a program have been generating the same type of results). Neither of these actors is strictly necessary - their purpose could be achieved by multiple outputs and inputs carrying the same data. However, this would place an extra burden on the method programmer. Moreover, these actors can be implemented efficiently: because both can fire immediately, the overhead of checking for enablement can be avoided, and because their function is known to be small, the overhead of starting a new process or thread of control to execute them can be avoided. The replicator is not allowed to contain a method to preserve the latter efficiency gain.

The merge actor is not allowed to contain a method since it has a different firing rule to the other actors, and a more complex interface to a method would therefore be needed, to incorporate some mechanism for indicating which input contained a data item. Although some benefit could be seen from allowing merge actors to contain a method - the method could, for instance, have provided sorting functions - simplicity of the programming interface was seen as the overriding factor.

The source actor is included in the notation because of its special firing rule; it was a design criterion that MeDaL graphs should express an entire program, from start to finish, including its initial input; therefore, there was a need for an actor which could fire without previous input. Additionally, method source actors can provide input data (including structured data such as in the matrix-multiplication example in section 3.4) to other actors.

The sink actor might be seen as being of limited use, since it produces no output, but it is included for two reasons. The first reason is completeness, since it complements the source actor; but the other is for reasons of modularity and code re-use. It is a principle of code re-use that programs should be structured as general-purpose modules; the MeDaL equivalent is to design actors with general functions. Such actors can then simply be plugged into a graph where their function is needed. However, if a particular output of that actor is not needed in some circumstances, it cannot simply be left unconnected, since all datapaths must connect an output to an input in a legal MeDaL program. The alternatives are either to re-write the actor code for those specific circumstances, or to connect that output to a sink actor which will simply ignore it. The latter approach has the extra advantage that if the program is modified later and that output is needed, the sink actor can then be removed and other actors inserted, without resorting to changes at the method code level.

The provision of specialised functions such as the merge actor may seem out of place in a medium-grained programming language; however, it is *because* these specialised functions are recognised primitives of the language, that they can be recognised at compile time and hence implemented as efficiently as possible. There is a tradeoff to be made between efficiency and generality, and in providing a few specialised actors, the MeDaL notation is an attempt to balance these factors.

The provision of the standard general-purpose method actor is obvious at the medium-grained level, being a vehicle for the programmer's code. The inclusion of the deep actor involves more complex issues. Since copies of its method code can execute concurrently on different sets of input data, it provides an extra type of concurrency available to a programmer using MeDaL.

Thus, there are three types of parallelism available in MeDaL. These can be characterised as:

- vertical parallelism - when two actors have a direct dependency (i.e. datapath) between them, they are normally drawn one above the other;
- horizontal parallelism - when two actors do not have any direct dependency, they can be drawn next to each other horizontally; and
- depth parallelism - a third dimension is needed for copies of the same actor, occupying the same place on the graph, but operating on different

data. Hence the name "deep actor".

Vertical, horizontal and depth parallelism correspond exactly to the structure, result and activity parallelism described in Chapter 2. This makes the maximum possible flexibility of programming techniques available when using MeDaL to design parallel programs.

3.3.3 Datapath Semantics

As described above, two types of datapaths are provided: the normal E-type and the specialised F-type. The main advantage of the F-type path is that when one data item needs to be used repeatedly, it need not be repeatedly transmitted by the producer. Obviously transmission of a data item on a datapath would incur some overhead, both in terms of processing time (consumed by the actor doing the transmitting) and of memory space (occupied by the datapath queue). This overhead would very likely be significant if the two actors involved are executing on different processing nodes of a message-passing architecture, especially if the data item was a large, structured data type object such as a matrix. Since MeDaL was designed with efficient implementation on both shared-memory and distributed-memory architectures in mind, this was an important consideration.

A corollary of this is that the F-type path allows different input streams to one actor to contain different numbers of data items. Using only E-types, to fire N times, an actor must have N items of data on every input path. This is reasonable at the fine-grain level where the operations carried out by actors are mathematical and have a set number of parameters. However, at a higher level, there is often no one-to-one relationship between streams of different types of data. When there is a one-to- N relationship, the one item would have to be transmitted N times, incurring the overhead described above.

A further use of the F-type path, in combination with the flush operation, is that it makes possible a "most recent value" service, for occasions on which an actor is only interested in the most recent value of a particular stream of data.

Thus, the F-type datapath is useful both in terms of functionality and potential efficiency. There were a number of alternative strategies which could have been used to provide a similar functionality, and these will be discussed in section 3.3.5. However, even accepting that the functionality of the F-type path is useful, this

does not justify the concept on which both types of datapath are based, namely the provision of a FIFO queue.

There are, of course, a number of alternatives to simple FIFO queues. Different types of dataflow system (see Chapter 2) have used a number of different types of path. In the feedback interpretation of dataflow, for example, a simple, finite FIFO queue is used. However, for MeDaL, it was decided that physical limitations for queues are an implementation issue which need not be incorporated in the abstract notation, hence the assumption of infinite-sized queues. In loop-unravelling dataflow, paths no longer implement simple queues; data tokens (items) may overtake each other according to information encoded in them at an earlier stage (using tags or colouring). The need for this arises because, in loop-unravelling dataflow, several instantiations of one actor can exist concurrently (in the same way as for MeDaL's deep actors). When this is the case, one instantiation may be started after another, but may (depending on the data values being processed) finish first; thus the data being processed become disordered. Because the actors are simple fine-grained operations, the actor receiving the results from the multiply-instantiated actor cannot in general be sophisticated enough to return the data tokens to their correct order, so this is done by the system, by re-ordering data according to its colour or tag.

However, because MeDaL is aimed at medium-grained processing rather than fine-grained, it is reasonable to assume that where a deep actor is used, *if* the order of the data is important, the receiving actor can include code to re-order the data items correctly. This gives the programmer flexibility to decide what mechanism to use for re-ordering; for instance, a tag field could be used if the data items are records. In the case of the matrix multiplication example in section 3.4, the first element of a fixed-size array can be used to indicate which row (of the matrix to which it belongs) a particular item represents. Since MeDaL also provides a persistent memory mechanism (see below), in which a re-ordered data structure can be built up over a number of firings, this is a reasonable burden to place on the programmer, and has the benefit of making MeDaL itself simpler and more elegant.

3.3.4 Persistent Memory

It has long been a principle of dataflow languages that dataflow operations themselves should not be able to retain any state, partly to keep intact the prized "freedom from side-effects" principle. On the other hand, the usefulness of some sense of persistent state has been recognised by some, leading for instance to decisions to allow cycles in dataflow graphs, as described in Chapter 2.

Absence of persistent state is also desirable from the point of view of the implementor of a dataflow language. This is because in the general case, if an actor fires, reserves some memory for its persistent state, then does not have enough input data to fire again, there is no way for the system to tell whether that actor will ever fire again; so the memory it reserved can never be reclaimed by a "garbage collector".

However, when considering medium-grained dataflow, there is one criterion which was seen as essential - that of ensuring that data structures are sufficiently large-grained to ensure efficient execution. In a medium-grained environment, it is important to be able not only to break data structures down into their components for processing, but also to be able to build small components up into larger ones. Without this ability, data structures can only get smaller, causing actors receiving many small data items to fire once for the arrival of each one, potentially causing unacceptable overheads. This problem is unique to medium-grained dataflow, since in fine-grained dataflow there is less need to build up complex data structures, and at the very coarse-grained level other persistent storage (such as filestore) can be used, since its overheads are small compared to those of communication between nodes.

The question therefore arises of what mechanism can be provided for persistent storage in which complex data types can be built up out of simpler ones - and should it be purely at the method level or should it be represented at the visual (graph) level as well? The following sections examine three possible answers to this question.

i. Memory Entities

One possible solution to the problem of persistent storage would be to provide a new, third type of entity within the MeDaL notation (in addition to datapaths and

actors) - a memory entity. An actor could, for instance, communicate with a memory entity to store data during one execution, and again to retrieve it following a later firing. Since such communication would be two-way, either two standard datapaths would be needed, or a new type of communication path could be employed. Figure 3.3a illustrates a possible visual representation of the latter.

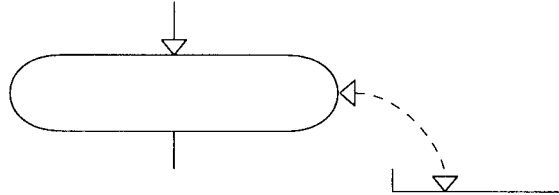


Figure 3.3a: Memory entity

The advantage of a notation like this is that it provides an explicit visual representation for the persistent memory, making it easy to see which actors in a MeDaL program use persistent memory. It is desirable that an actor which needs persistent memory should be able to store data items of more than one different data type; this could be accommodated by allowing connections to several typed (single data type) memory entities, several typed connections to one general (non-type specific) memory entity, or one typeless connection to a typeless memory entity. In the latter case, the send and fetch operations needed by the actor could even be pattern-based operations similar to Linda's tuple-space operations (see Chapter 2). Since the contents of the memory entity would be managed by the MeDaL run-time system, garbage collection would be possible.

There are, however, a number of disadvantages. The extra visual representation would make the MeDaL notation more complex, and hence less easy to understand. In particular, the introduction of a new type of connection which does not behave like normal datapaths is confusing and counter-intuitive. The method programming interface, too, would have to be complex, specifying what type of data was required from the memory entity (or which memory entity to query). Each fetch operation from the memory entity would require two messages to be passed (one in each direction), and even if this did not constitute a major communications overhead, it would require some programming complexity. However, the major objection is perhaps that this mechanism does not fit in well with the dataflow paradigm of data flowing from one actor to the next. A "cleaner" solution would incorporate

persistent memory into either the actors or the datapaths, and these are the remaining two options to be examined.

ii. Persistent Memory in Actors

It was noted in the introduction to this section that allowing actors to own persistent memory in which they can store some state between firings is a disadvantage from the point of view of automatic garbage collection. However, this approach does have a number of merits which weigh against this. First among them is simplicity: each actor with persistent memory could simply own a typeless pointer to a block of memory (which could, on request, be extended). The actor could then manage the storage of variables within this memory, so the mechanisms used could be as simple (or complex) as the application required. The presence of persistent memory with the actor could easily be denoted visually, for instance by shading one end as in Figure 3.3b; a visual distinction would be useful to the MeDaL programmer if only to see at a glance which actors owned persistent memory.

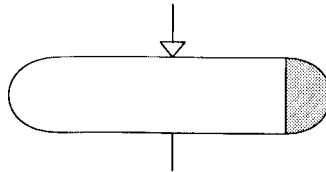


Figure 3.3b: Persistent memory within an actor

This mechanism provides persistent memory in a very simple and natural way, given that the requirement is for an actor to be able to own some persistent memory in which, for instance, simple or small data types can be built up into more complex ones. The programming interface need not be particularly complicated. The only disadvantage is that the onus of storage management (for instance, indexing what items of data are where in persistent memory, as well as the release of redundant memory) is on the method programmer.

iii. Persistent Memory in Datapaths

The final option, and that which was chosen for MeDaL, avoids the latter disadvantage. As described in section 3.2, the approach chosen consists of extending the semantics of the "send" operation on output datapaths to include a variant which uses the datapath being sent to as persistent memory (i.e. `send-sticky`). This is extremely simple and has a number of advantages, as follows.

Firstly, datapaths are already ordered and typed, so no indexing method is needed to keep track of what data items of what data type are where in the persistent memory area, as would be needed in the other schemes above. Secondly, memory management of the datapaths is already done by the system, so persistent memory can be managed by the same mechanism, keeping both the implementation and the semantics of the persistent memory mechanism simple. Thirdly, no explicit retrieval operation is needed, since it can happen automatically at fire-time.

The only slight disadvantage is that the use of this mechanism is not obvious at the MeDaL graph level, since it is implemented at the method interface level, in the form of operations on datapaths by actors - hence it does not have a visual representation. However, its use could easily be indicated by a run-time or postmortem tool, by changing the colour or line style of a datapath to indicate that it contains persistent data.

3.3.5 Multiple Streams

The problem arising when an actor needs to have different numbers of data items arriving on different input streams was described in section 3.3.3 on datapath semantics. This problem is closely related to the question of how to manage persistent memory, because if a mechanism is to be provided by which one data item is retained from one stream while many arrive on another, then clearly one approach is to store that one data item in persistent memory.

The F-type paths described in section 3.3.3 are in effect a form of persistent memory, retaining the last item in a FIFO queue to stop it becoming empty. In effect, this is a method of altering the synchronisation of data arriving at actors' inputs without altering the firing rule. Its advantage is simplicity (indeed, it is invisible at the method code level); its disadvantage is that it forms a second type of persistent memory to that described above.

In designing MeDaL, a number of options were considered which would have solved the multiple-streams problem and the persistent memory problem in a more integrated way. This section describes seven of these options, and why only the last of them was chosen.

i. The Do-nothing Approach

Provision for multiple streams and persistent memory is not strictly necessary if

actors' outputs are fed back into their inputs. Figure 3.3c illustrates this being done. A merge actor is used so that the initial input and subsequent outputs arrive at the same method actor input.

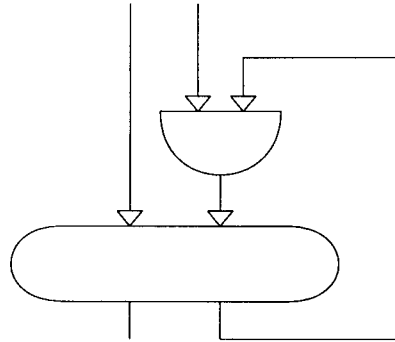


Figure 3.3c: Output looped back to input

The disadvantages of such an approach are many. For instance, the graph surrounding this one actor would become extremely complex if it had, say, five outputs which might either be looped back or (when finished with) delivered to somewhere else. Also, this approach allows an N:1 relationship between input streams, but not N:M. The loop-back circuit cannot easily be flushed, to allow replacement of the value being used on that input to the actor. The overhead of transmitting data on an output path and storing it until the next firing would be incurred, even if the next firing was immediate. And if complex data types were being built up from simpler ones during the cycling process, data typing of datapaths would have to be relaxed and the method code programmer would have the extra burden of working out what stage of the process had been reached for a particular input.

ii. A New Method Merge

This idea combines the MeDaL merge actor with some form of persistent memory. Relaxing the rule that merge actors cannot contain methods, this actor would contain a method which, according to the merge actor's firing rule, would execute whenever data arrives on any one of its inputs. It would then store this data in a persistent-memory queue, one for each output; and if there was data in every queue, would dispatch the head of each queue on its outputs. It would have semantics similar to those of the F-type path, retaining the head of the queue if the tail was empty. In this way, it would equalise the numbers of items of data on each datapath stream. Figure 3.3d illustrates such an actor with in-actor persistent memory, but it

could equally well make use of persistent memory in datapaths.

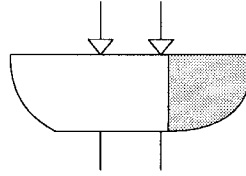


Figure 3.3d: Equalising method merge

The advantage of this idea is that only one type of datapath would be required (the simple E-type). However, this solution would not be efficient. The actor-firing overhead would be incurred each time data arrived on each path. The actor would be a potential bottleneck, since actors cannot fire again until they have finished executing; in this case, a bottleneck would occur if data items arrived on the input paths faster than the actor could process them. Finally, because data items would be duplicated on the output paths (for instance, in the case of two streams of N and 1 items of data, the one item of data on the second stream would be transmitted N times) far more datapath storage would be used, needlessly duplicating values.

iii. Datapaths Into Persistent Memory

Another strategy would be to adapt the ideas of persistent memory as being either within an actor, or a separate entity. If an actor's private persistent memory was considered separate from it (either attached or as a separate memory-entity), a system could be employed whereby datapaths could deliver data items direct to persistent memory rather than to the actor. The actor would still be subject to the strict enabling rule, but this would only apply to the actor's inputs not the persistent-memory inputs. Inputs to persistent memory would be automatically handled by the system when they arrived, by some kind of server mechanism. Possible visual representations of this are shown in Figure 3.3e.

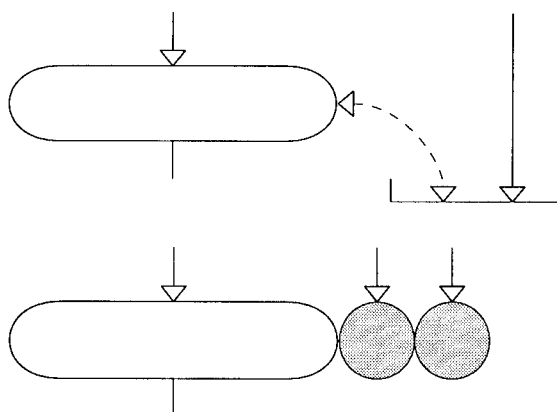


Figure 3.3e: Datapaths to persistent memory

The attraction of this method is that again, F-type paths are not needed since the same mechanism is used for general persistent memory, and storage of data from streams of potentially different numbers of items. However, for user-defined data types to be transmitted through datapaths, the server code and the method interface with persistent memory would need to be sophisticated. Moreover, the firing rule might need to be altered to ensure that the actor could not fire until at least one data item had arrived in each area of persistent memory, since that item might be essential to execution. Finally, the actor would not be able to fire when data arrived only at a persistent memory server and not at the actor's main inputs. This mechanism, therefore, is not very flexible.

iv. Using Shared Memory

The last problem, that of not firing even when some data had arrived, would be solved if actors could share persistent memory. All datapaths would go to actors, and the actors would be subject to the normal firing rule. If actors could share persistent memory stores, they would not need to be synchronised; yet every time any item of data arrived, an actor would fire and deal with it as necessary. Possible representations for this, using a separate memory entity and using joint access to internal persistent memory, are given in Figure 3.3f. In any representation, the graph would become very complex if more than two actors were to share persistent memory.

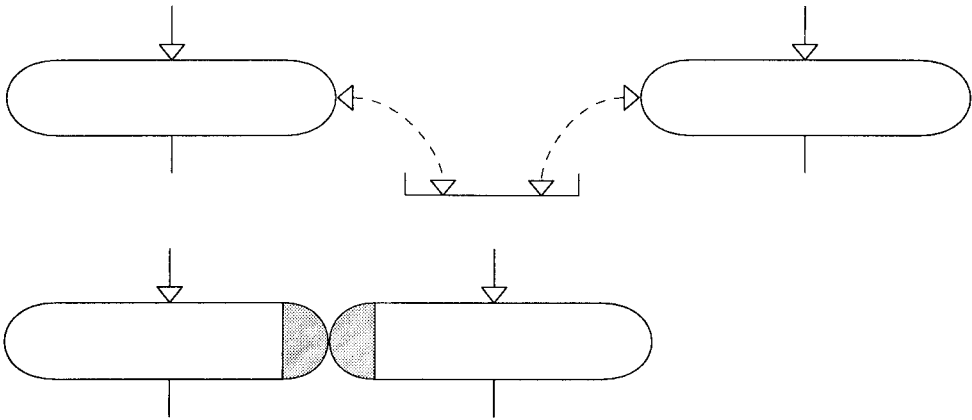


Figure 3.3f: Shared memory

Although this approach solves the problem of different numbers of data items arriving on different streams, it does not solve that of an actor firing only to discover that a vital data item from a different stream has not yet arrived. Indeed, this is the kind of synchronisation problem which dataflow's strict enabling rule is designed to solve, so it would clearly be counter-productive to re-introduce such problems in this way. Moreover, if the function of some actors is purely to accept data and store them in persistent memory, this method is perhaps rather heavy-handed, incurring the actor firing overhead each time data arrives on any stream. Meanwhile, the prize of actors' freedom from side-effects is lost.

v. Use of Demand Dataflow

An altogether different strategy would be to incorporate demand-driven dataflow into MeDaL, which is otherwise purely data-driven. An actor would fire when all "data-driven" inputs had arrived; it would then "demand" data from its other inputs, which would then be provided by actors above it, which would keep the items in persistent memory until needed. Figure 3.3g illustrates a graphical representation for this, using two different styles of arrowhead for data-driven and demand-driven paths. The demand-driven path has arrowheads at both ends to emphasise that messages are passed in both directions (the demand upwards, and the data downwards).

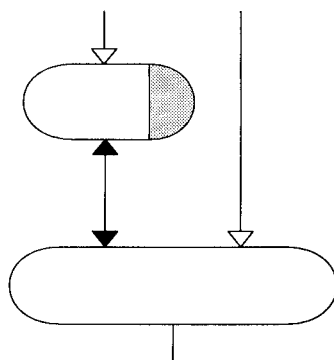


Figure 3.3g: Demand-driven dataflow

Of course, the fact that two messages would need to be sent for each demand input would decrease efficiency on a distributed architecture; streams with differing numbers of data items would be catered for, but only at the expense of transmitting some items across datapaths more often than necessary, as with the method merge solution above. In addition to this, both the method programming interface and the visual language would be made more complicated by this extension. The desire for both efficiency and as simple as possible a programming model ruled this idea out.

vi. Synchronous and Asynchronous Inputs

The final approach to be considered involves persistent memory and a more complex firing rule. Most of the previous solutions attempted to keep the model simple by maintaining a simple firing rule, but all involved other complexities.

However, the firing rule should not be considered sacrosanct. Rather than insisting all inputs contain data before an actor can fire, it would be possible to have a system in which not all inputs need be. Of course, some inputs might be vital prerequisites to an actor's execution, but others might be optional. The necessary ones could be called **synchronous** inputs, and the optional ones **asynchronous**, since only the necessary ones need be synchronised (by all being present) for the actor to fire. The two types would be distinguished visually; Figure 3.3h illustrates a distinction either by arrowhead style, or by their arriving from different directions: synchronous inputs arriving at the north edge of the actor's box, asynchronous inputs arriving at the west edge.

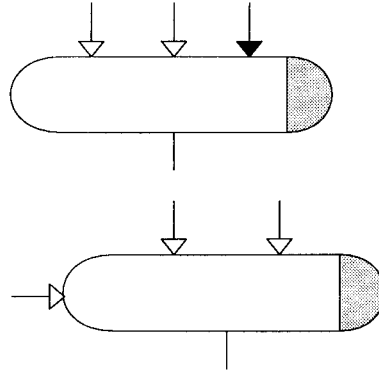


Figure 3.3h: Synchronous and Asynchronous inputs

The actors in Figure 3.3h are shown as containing persistent memory. This would be necessary in order to cope with different input streams containing different numbers of items, since the smaller number could be stored in persistent memory. Thus, this scheme would be relatively efficient, requiring no extra transmission on datapaths of data items needed more than once. Persistent memory in datapaths rather than in actors could equally well be used.

The only disadvantage of this scheme is that the method programming interface would need to be somewhat more complex, since different numbers of inputs may be present at each different firing. The interface with the actor method would need to include information about which inputs were present and which were not, and this information would have to be decoded by the method.

vii. E-type and F-type Paths

The E-type and F-type datapaths described in section 3.2 are very similar to the synchronous and asynchronous paths described above. However, in the case of F-type paths, the normal firing rule is circumvented by the datapath delivery mechanism itself, rather than being explicitly relaxed. This has the advantage that it can be specified purely at the visual MeDaL level rather than requiring the method interface to be made more complex. In conjunction with persistent memory being located in datapaths rather than in actors, it is as efficient as synchronous and asynchronous inputs, while being simpler for the programmer to use. Hence, out of all the alternatives described above, the E-type and F-type datapath solution together with datapath-based persistent memory was chosen as being the best solution to the problem of how to provide efficient persistent memory.

3.4 Examples

Having presented the MeDaL notation, this section provides some examples of its use. The main example used here is a parallel matrix multiplication program. The features of the notation which it illustrates are described.

First, however, the MeDaL library must be described. It was stated in section 3.1 that a library of actors would be provided for the programmer to make use of, and a minimal library is described in the following section. These library actors are an integral part of the MeDaL language; however, they use actors of types described above and do not introduce new features of syntax or semantics.

3.4.1 The MeDaL Library

This library contains five method actors, based on the primitive types described above, which in an implementation would contain instructions to the MeDaL run-time system. The reason for their existence is to provide a clean interface between the MeDaL program and the "outside world" i.e. the host computer system.

Standard Input and Output Actors

The standard input actor (labelled *stdin*) is essentially a source actor which fires whenever a line of character input appears on the program's standard input stream, as provided by the host operating system. The data type of the output path is text-string.

The standard output actor (*stdout*) is the corresponding sink actor, sending its input to the program's standard output stream. Again, the data type of its input must be text-string. (In figure 3.4a they are illustrated with their input or output path present.)

The terminology for these actors is inspired by the Unix operating system, which provides these streams as a standard facility which programs can use without needing knowledge of whether, for instance, the input comes from a keyboard, file, or other device. However, MeDaL does not assume the Unix model and the *stdin* and *stdout* actors could be implemented to access devices directly on other systems; they are intended to provide portable input/output mechanisms.

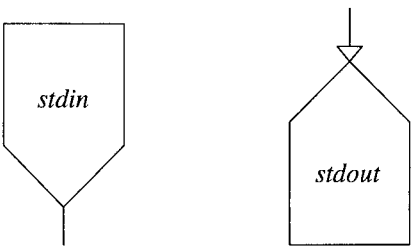


Figure 3.4a: Standard input and output actors

File Input and Output Actors

These are based on general-purpose actors. The file input actor *filein* has one input which specifies the filename, and two outputs, one for any error messages, and one on which the contents of the file are placed, line by line. All three datapaths are of type text-string. The fact that the filename to be used is an input to the actor means that the filename is under program control at run-time.

The file output actor *fileout* has two inputs and one output: one input for the filename, and the other for lines of text to be sent to the file; the output is for any error messages. Again, the inputs and output are all of type text-string. In figure 3.4b the *fileout* actor is shown with an F-type path as the filename input, with the effect that the filename to be used need only be transmitted once, and will remain the same each time a new line arrives on the data input.

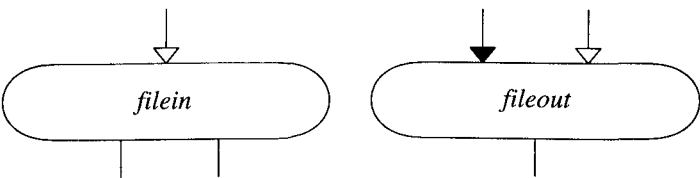


Figure 3.4b: File input and output actors

Halt Actor

The halt actor is a sink actor (labelled *halt*) which contains a method. When executed, its effect is to cause the entire program to halt as soon as all currently executing actors have finished their processing and terminated (it was stated in section 3.2.2 that all actors are assumed to terminate). This allows a more elegant termination than waiting until all actors run out of inputs. It is intended that it could

also be used for debugging, not only in forcing early termination of the program as a whole (i.e. while there are still enabled actors) but for use as a breakpoint after which the program could be restarted.

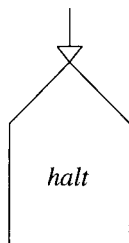


Figure 3.4c: Halt actor

3.4.2 Example MeDaL Program

As a simple example of how a program can be expressed in the MeDaL notation, consider a program for multiplying two matrices together. It is well established that there is potential parallelism in this computation, since each element of the resulting matrix depends only on the values of one row of the first matrix and one column of the second. Assuming the first matrix A is of size i,j and the second B of size j,k , the result will of course be of size i,k . Thus, for maximum parallelism, one would create $i \times k$ processes, each of which would compute one element of the result (by multiplying one row of A by one column of B ; a vector-by-vector calculation).

However, this is a fine-grained computation (each process computing one value). On a MIMD multiprocessor, where the process creation overhead is significant, such an approach is likely to be inefficient. A better solution at the medium-grained level is to create k processes, each of which computes a whole column of the result matrix, using the whole of A and one column of B . In other words, this is a matrix-by-vector calculation. Figure 3.4d below is an example MeDaL program, *matrix-mult*, which takes this approach. The actor code associated with this MeDaL graph can be found in Appendix B.

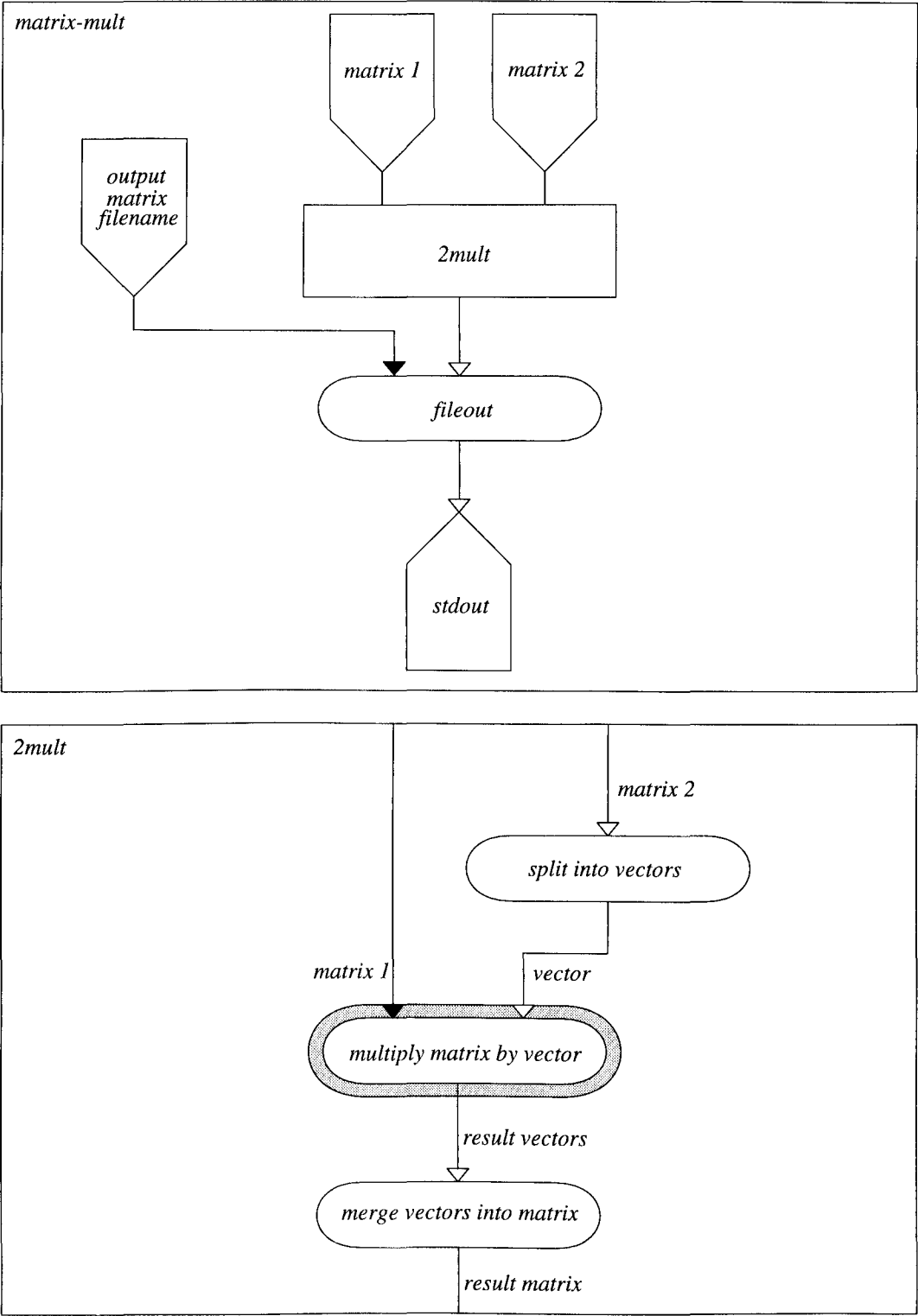


Figure 3.4d: MeDaL example program "matrix-mult"

The program consists of two companies, the root company (labelled in the northwest corner with the name of the program) and a company called *2mult* which serves to group together the actors concerned with the main processing work of the

program. *2mult* can be seen in its "component" form within the root company, and in its expanded form below. It can be seen that the program could be rewritten to eliminate *2mult*, by placing its contents where the *2mult* component appears in the root company. Indeed, this is effectively what happens at run-time; the use of a company in this case is only for notational convenience.

The root company comprises three source actors, one library actor and one sink actor as well as the *2mult* company. The source actors *Matrix 1* and *Matrix 2* each contain a simple method which simply transmits a matrix on the output path. The other source actor transmits a single character string on its output path, containing the file name to be used for storing the matrix resulting from the computation. The *stdout* actor exists purely to print out any error messages which may be output from the *fileout* actor. This illustrates the building-block use of the sink actor; although in this case the error output of *fileout* is not dealt with by the program, *fileout* need not be rewritten so as not to produce one; it can be used as it is. Additionally, the program could later be expanded to handle errors, simply by replacing the sink actor with a general-purpose type actor.

The *fileout* and *stdout* actors are used as examples of library actors; in a real program, it is likely that *filein* actors would also be used to load in the input matrices. They are omitted here only for simplicity.

The expanded form of the *2mult* company illustrates how companies other than the root company receive inputs from "outside" (the paths which appear from the north side of the box) and can transmit a result back "outside" (the path which ends on the south side of the box). In this case, the two inputs are the matrices to be multiplied. The second of these goes to the actor labelled *split into vectors* which fires only once, but transmits each row of that matrix on its output as a separate vector.

The *multiply matrix by vector* actor fires once each time one of these vectors arrives; it has two input paths, one for a matrix and one for a vector. Because the matrix input is an F-type path, the matrix is not consumed and remains present as an input each time this actor fires. Moreover, since *multiply matrix by vector* is a deep actor, each matrix by vector calculation can be computed in parallel (assuming sufficient processors are available). Each result (a column of the result matrix) is then output.

The *merge vectors into matrix* actor fires once for each time one of these result columns arrives. Its only purpose is to copy the vector to the appropriate column of the result matrix, which is built up (in textual form) on its output path using the *send-sticky* technique. When every column is present, it then sends the whole resulting matrix. This goes to the *fileout* actor which stores it in a file, and the program terminates.

Of course the multiplying actor and the merging actor need to know *which* column of the result their vector input represents. For each vector, an index value is needed to indicate which column the vector is. This index could be passed from *split into vectors* to *multiply matrix by vector* on an extra datapath, then on from *multiply matrix by vector* to *merge vectors into matrix* on another separate datapath. This would have the advantage of making the passing of this piece of information explicit at the MeDaL graph level. However, it also carries the penalty of transmitting and storing a separate item of data. Since this index is a numerical value, and the vector also contains numerics, it is more efficient to transmit it as part of the vector, for instance as the first element of each vector at each stage. In the interests of efficiency and simplifying this example graph, this approach was chosen here. If the index value was not of the same data type, it could at least be sent in the same data item (using a structured data item).

Two types of parallelism are illustrated by this example: structure (pipeline) parallelism, since the actions of splitting a matrix into vectors, multiplying, and merging the results can be done in parallel; and activity parallelism, due to making a number of instantiations of the actor which does the multiplication. An alternate strategy would have been to control the amount of parallelism by constructing the graph in the form of a number of side-by-side copies of the multiplication actor, each with its own datapath from the vector producer and to the vector merger. In other words, to employ horizontal rather than depth parallelism in designing the program. While this is a valid approach, it would be less flexible in this particular example, since the amount of parallelism available to be exploited by the program would then be fixed in the design, rather than decided at run-time, making the program harder to modify later.

3.5 Summary

The example above has been described in detail, and if this explanation is added to the size of the MeDaL graph and its method code, then the simple matrix-mult program may not seem very concise. However, this explanation is provided only to clarify the description in the previous section of the operation of programs expressed in MeDaL. Once one fully understands how MeDaL programs work, MeDaL graphs and method code need very little annotation, and provide a concise description of a potentially parallel program. This is important because naturally, the easier it is to understand (and modify) the design of the program, the easier development and maintenance of the software becomes.

The main advantage of MeDaL is in the implicit parallelism found in dataflow graphs. In addition, MeDaL provides a number of specialised features. To summarise these:

- **MeDaL programs consist of two levels: a dataflow graph and method code**

This scheme is of course designed primarily for medium-grained parallelism, where chunks of code the size of imperative language procedures or functions can efficiently be executed in parallel. Rather than hindering understanding of the program, the two-level structure of MeDaL programs (graph and textual code) is designed to make programs easier to understand, describing first the whole structure - the graph level - and then, once that is understood, the computational details. Another benefit of this structure is that existing imperative language functions can be adapted and placed in a MeDaL "harness" graph to facilitate their transformation into a parallel program.

- **MeDaL graphs consist of actors and datapaths**

Actors fire according to well-defined firing rules, and data flows between them on unidirectional datapaths. These concepts are simple, making the graphs easy to interpret. Data flows from top to bottom, but cycles are allowed, for flexibility.

- **MeDaL provides six primitive actor types**

There are six basic types of actor: the general-purpose actor, the deep actor, the source and sink actors, the replicator and the merge actor. Each varies in its firing rule or number of outputs. All but the replicator and merge can contain a method,

i.e. a section of code which performs some function, expressed for instance in some textual imperative language. The non-method actors are more fine-grained, but are not inefficient since they can easily be optimised out at compile-time. The provision of six types of actors is a compromise: an attempt to provide flexibility and generality without making it necessary to learn the functions of a large number of different primitives.

- **Datapaths can carry structured data**

MeDaL's datapaths can carry structured data items such as matrices, vectors, records etc, corresponding to the input parameters of the method functions, and is hence efficient at the medium-grained level. The standard datapath is the E-type which represents a simple FIFO queue, a simple but general concept.

- **Datapath semantics provide two types of persistent memory**

Memory which is persistent between different firings of an actor is desirable for two reasons: it allows actors to have two or more input streams which receive different numbers of data items, and it allows actors to build up complex data items out of simpler ones over a number of firings. MeDaL datapaths provide a different type of persistent memory for each of these two purposes. Firstly, at the graph level, F-type paths are provided which are FIFO except in that they retain their last element rather than becoming empty. Thus, actors do not have to wait for the same number of input data items on every path. Secondly, the semantics of the operations which actors perform on datapaths include not only a normal send, but a send-sticky which does not allow the data item being "sent" to be delivered, so that it is still available on the next firing. Thus output data can remain "stuck" in an output path to be manipulated by the actor using more than one set of input data.

The provision of the features described above are designed to make MeDaL a useful language for the design, implementation and testing of parallel programs for medium-grained multiprocessors. This chapter has described the language itself, and illustrated its use in designing a parallel matrix multiplication program. The next chapter, therefore, turns to the implementation of MeDaL as a design and implementation language.

Implementation Issues

In order to implement MeDaL as a complete programming system, with the features described in the previous chapter, a number of distinct modular tasks can easily be identified. A graphical editor is needed to edit the MeDaL graph; and a text editor is needed to edit the actor methods. A module is needed to extract information from the MeDaL graph to form the basis of a parallel program (the "harness"), transforming the MeDaL program into a structure appropriate to the target architecture. Another module may be needed to package the method code into a form which can be directly executed, for instance by adding code which fetches data items from datapaths and passes them as parameters to function calls. The harness, and the code which it surrounds, can then be combined into a parallel program using an existing compiler. Since MeDaL allows run-time expansion (of datapath queues and of the MeDaL graph itself, through the use of companies) a run-time library module would also be needed to support or manage execution of the parallel program. Figure 4.0a illustrates the interconnection of these modules.

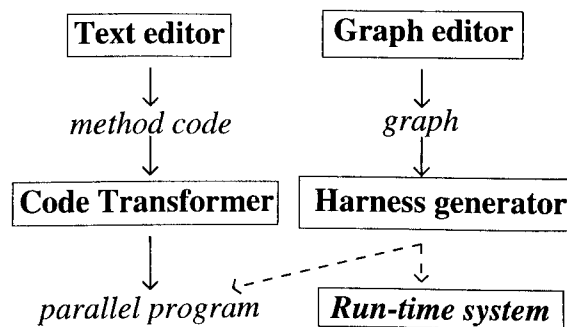


Figure 4.0a: Modules of a MeDaL programming system

The dotted lines from the harness to the program and the run-time system indicate that harness information could be built into either the program or the run-time system; there are a number of options for the exact constitution of the run-time system, and these are enumerated in the next section.

The technology involved in text editors is essentially very simple, and graph

editors are also well understood (see Chapter 2). The MeDaL graph editor would simply allow the addition, movement and deletion of MeDaL actors and datapaths, according to the syntax rules described in Chapter 3. It would allow "browsing" of the graph, and would need to support examination of every level of the graph (for instance by opening further graph-editor windows onto companies, and by opening text-editor windows onto method code associated with a particular actor). These operations are sufficiently simple not to require further discussion here; therefore only the final *output* of the graph editor, which is the input to the harness-generating module, will be considered here.

This chapter concentrates on the other rôles of the MeDaL programming system: how parallel code can be automatically generated from actor methods and the MeDaL graph, and how to support the execution of this parallel code at run-time. Section 4.1 discusses in more detail what is needed from the various parts of the programming system; section 4.2 describes the programming interface between the system and the programmer's actor method code; section 4.3 raises some of the issues of how such a system can be implemented on a distributed-memory architecture; and section 4.4 describes the implementation of the system on a shared memory architecture. This implementation was used for the experimentation to be described in the next chapter. The machines used for the work described in this chapter were an Intel iPSC/2 hypercube (distributed memory) and an Encore Multimax 520 (shared memory).

4.1 Functions of the Programming System

This section examines the techniques which can be used to generate executable parallel programs from method code and MeDaL graphs, using graph and method transforming modules and a run-time system, as described above. First, however, it is necessary to define more clearly the distinction between these transformation modules and the run-time system, in terms of the work which they do.

4.1.1 The Division of Work

As mentioned in the previous section, there are several possible strategies for the provision of a run-time system, depending on whether harness information is incorporated or not. The options are:

- a standard library of support functions could be provided;
- a tailor-made run-time system could be built for each application using information from the harness;
- a hybrid approach could be adopted, with a library of generalised functions which is fed data about the dependencies of a specific application, for instance by means of a file loaded before the start of dataflow execution.

The choice between these options is simply one of work allocation between the modules of the MeDaL programming system; the greater the support given by the run-time system, the less work needs to be done by the transforming modules to turn the MeDaL graph and method codes into an executable program.

Each approach has its advantages and disadvantages. The provision of a generalised, pre-compiled run-time library is a common approach, and has the advantage that it reduces compile time. However, more coding effort must be expended in ensuring that it is generalised enough to deal with all possible requirements, and will result in functions which contain code not all of which is used in every situation. This is a disadvantage on distributed-memory architectures, since it is desirable to avoid the overhead of distributing unused code between processing nodes and storing it in the relatively limited memory there.

The tailoring of the run-time system effectively moves complexity from the run-time library to the transformation modules, but causes longer compile-times. Finally, the hybrid approach, while offering a compromise, has the unfortunate characteristic that the reading-in and interpretation of the application-specific information adds to the total run time of the program. The obvious comparison which can be drawn is with compiled versus interpreted languages - interpreters can be very flexible but are, in general, slower. For this reason, this approach was not chosen.

The criterion used to choose between the other two approaches was that of the characteristics of the underlying architecture. On a distributed-memory architecture, relatively complex decisions must be made about the way in which processing tasks will be allocated to processors, in order to minimise communication (which is the major factor determining throughput). On a symmetric shared-memory multiprocessor, such decisions are not necessary. Since these decisions take time, it

is desirable to take them at compile-time, to ensure the best possible efficiency of execution (more detail about the nature of these decisions is given in the next section). Therefore, the second approach (with the bulk of the work done in the transformer modules) was adopted for use with the distributed-memory architecture, while the first approach (with a pre-compiled, generalised run-time library) was used on the shared-memory architecture, bringing the benefit of faster compilation without a significant loss in execution efficiency.

Thus, two distinct balances of work allocation between modules of the system were tried: on the shared-memory architecture, simple transformation modules generated a program which executed with the aid of a relatively complex run-time library; and on the distributed-memory machine, more complex transformation modules were planned, supported by a simpler core run-time system. Section 4.3 describes the experimentation using the distributed-memory machine, and section 4.4 the implementation using shared-memory. However, there are some general issues involved which are common to both, and these are considered in the immediately following sections. To summarise, the three components of the MeDaL programming system to be considered here are:

- the method code transformer
(input: a collection of fragments of HLL code, the actor methods)
- the harness generator
(input: some representation of the MeDaL graph)
- the run-time system
(possible input: the harness)

Each of these will now be dealt with in turn.

4.1.2 The Method Code Transformer

Ideally, the HLL method code written by the programmer should not contain any code specifically to perform dataflow functions, in order to minimise programming effort and to maximise portability. The input data (data items from the input paths) and output data (items to be transmitted on output paths) should be passed in and out using the programming language's normal mechanisms for parameter passing. The method code transformer, therefore, must undertake the task of interfacing actor code with the mechanism used to implement datapaths between actors (through shared memory or inter-node communication links as appropriate). The

main function of this interface is to extract information about what inputs and outputs an actor has, and binding these to appropriate calls to the datapath mechanism.

The simplest way of implementing such an interface is to generate a "wrapper" function for each actor, which undertakes the task of retrieving the input data from datapaths, then calling the method code, passing in the data thus retrieved as input parameters. It is this wrapper which is called when the actor fires, rather than the method code itself. Thus, a program comprising the method functions and the wrapper for each one is the output of the method code transforming module. The requirements for the programming language interface (between the method code and the wrapper and/or the run-time system) are discussed further in section 4.2.

4.1.3 The Harness Generator

While the method code is transformed into code corresponding to the program's actors, the MeDaL graph must be used to generate code corresponding to the program's datapaths (the "harness" which allows dataflow-based execution). At first glance, the harness generator's task of transforming lines in a graphical editor into executable code might seem a much bigger task than that of the method code transformer which merely augments HLL code sections. However, since the main function of the datapaths is to transport data, and since most commercially available multiprocessors provide mechanisms which do this, the task is not as great as it might seem. Only code which makes use of these mechanisms in the appropriate way need be generated.

Therefore, the task of the graph transformer is to extract information from the graph concerning the source and destination actors of each datapath, and to generate efficient code to enable the method wrappers and the run-time system to control the available data transport mechanism. This code is no more complex than a series of assignments mapping datapaths to actor inputs, and actor outputs to datapaths.

While doing this, the graph transformer must take into account actors which do not contain a method. Although these are *logically* actors, the fact that they contain no method means that they do not need a method wrapper, and in fact do not need to be executed separately at all; so it is reasonable to consider them merely part of

the graph for the purposes of generating executable code. An example would be a merge actor: suppose actors A and B have output datapaths x and y respectively which are the inputs of a merge actor C , and that the output datapath of C , called z , is the input of actor D . Rather than generating code which maps $A \rightarrow x$, $x \rightarrow C$, $B \rightarrow y$, $y \rightarrow C$, $C \rightarrow z$ and $z \rightarrow D$, it would be more efficient simply to map $A \rightarrow z$, $B \rightarrow z$ and $z \rightarrow D$. In this way, a non-method actor and two datapaths can be optimised out completely. Other optimisations are similarly possible when generating the datapath code for other non-method actors.

The code generated in the way described above is then incorporated either into the run-time system, or into a further level of wrapper around the method code, according to the approach being adopted (see above). In the case in which the code derived from the MeDaL graph is used in a new wrapper level, a wrapper is created round each company giving details of the connections between actors in the company. This code is not executed like an actor, but is called by the run-time system whenever data first enters a company, to find out which actor the data should be delivered to.

The harness generator would also need to generate code to handle such tasks as type conversions in a heterogeneous computing environment. However, for simplicity's sake, this thesis only considers homogeneous multiprocessors, i.e. those in which the CPU and data representation are the same on all processing nodes.

4.1.4 The Run-time System

Having considered the code transformer and the harness generator, the remaining component is the run-time system, and much of the functionality of this has already been implied. The full list of the run-time system's functions is as follows. It must:

- do any machine-specific preparations for parallel processing;
- maintain a "roadmap" describing datapath connectivity (see below);
- generate any data from MeDaL source actors;
- implement MeDaL library actors such as *halt* and file input/output;
- decide when method actors are able to fire, and invoke them via their wrappers;
- detect termination of the program and shut down gracefully.

In addition, it may perform other functions such as the saving of "checkpoint" snapshots of the running program enabling it to be restarted later from the point at which it was saved; the collection of datapath contents and actor variables for debugging; and run-time garbage collection.

The **roadmap** referred to above is a dynamic data structure, the maintenance of which is central to the role of the run-time system. Essentially, it records the mapping between actor outputs and datapaths, and the destination of each datapath. It takes the form of a set or array of companies, each of which contain actors, datapaths and possibly companies. As described above, the code which sets up this data structure is either part of a tailor-made run-time system, or is available in code which can be called by the run-time system. However, because MeDaL graphs can expand at run-time (for instance through recursive calling of companies), the roadmap must be dynamically extensible.

The roadmap must also contain information such as the physical location in memory of datapaths and actors, and information relating the group of datapaths which form the inputs of each actor, so that the run-time system can easily check whether an actor is runnable (i.e., when all of such a group of datapaths are non-empty).

It is this run-time information held in the roadmap which is (indirectly) accessed by the method programming language interface, including the method wrappers, to put into effect the "transmitting" and "receiving" of data through datapaths. This interface is described next.

4.2 Programming Language Interface

It has already been stated that the code transformer module takes method code and generates code consisting of each method with a wrapper round it, to handle the receiving and sending of items of data from and to datapaths. It has been asserted that, ideally, the method code should not need to include any code to control the flow of its input and output data through the dataflow graph. However, there are a number of issues which mean that this idea may not be quite achievable in practice. This section identifies these issues, and describes how they can be tackled using FORTRAN, C and C++ as method implementation languages.

Although there are essentially only two points at which method code needs an interface to the system - the receiving of input data and the sending of output data - there are complications to both of these. Because of the semantics of datapaths, it is necessary for method code to have two operations on output data (`send` and `send-sticky`), and to be able to find out whether or not a particular output path currently contains a "sticky" item of data from a previous firing of that actor.

There are also a number of features which are desirable from the point of view of efficiency; for instance, if an input is to be passed on to an output unchanged (or largely unchanged, if it is a relatively large structure) physical copying of the data from an input parameter to an output parameter is undesirable. Since the aim of MeDaL is to provide a programming environment which is efficient at the medium-grained level, this type of issue must be addressed.

4.2.1 Programming actors in C and FORTRAN

Passing input parameters to an actor method is simple in any block-structured imperative language, since a mechanism already exists, namely that of input parameters to functions or procedures. Passing output parameters back to the caller is less easy in languages like C and FORTRAN, since these languages only have provision for functions to return one value as the result, whereas in MeDaL there can be an arbitrary number of outputs from an actor. An approach often adopted is to pass in (using the input parameter mechanism) pointers to the variables which will be used as the function's outputs. A simplistic solution to the need for "stickiness" semantics would be simply to pass in not only variables for the actor's outputs, but a flag for each output variable marking its "stickiness" which could be tested by the method code, and set or unset by it, resulting in a sticky or non-sticky send when the method terminated - the wrapper code handling the actual sending. Additionally, each output would need a further flag marking whether or not there actually was any data to be sent on that particular output (since it is not a requirement that data items be sent on every output for every firing).

Although this simplistic solution is easy to implement, it has two main disadvantages. Firstly, if the wrapper code is to handle the transmission of each output variable, only one item of data can be sent on each output path for each firing - unless a more complex scheme, such as using variable-length arrays for each potential output, is used. Secondly, if one particular output of an actor is ready

half-way through its execution, and the datapath on which it is to be sent is empty, not sending it until the end of execution this may cause the destination actor of that datapath to wait for that particular input longer than necessary. For optimum efficiency, it must be possible to send an item on a datapath as soon as it is ready to be sent.

The simplest way to solve these problems in C and FORTRAN is to use function calls representing the `send` and `send-sticky` operations. These may be calls direct to the run-time system, or more probably, to the wrapper level. This is likely to be necessary because, to transmit data on a datapath, the function needs not only the data item itself and information about which of the actor's output paths to use, but also the position of the actor in the roadmap - something which cannot be determined at compile-time. Since it is also desirable to hide this detail from the actor programmer, each actor wrapper could include a function called by the actor method code to fill in this information before the data-sending operation goes ahead. Alternatively, in C, macros could be provided instead of function calls, so that the appropriate code would be added to the main body of the actor method before compile-time (by the C preprocessor).

However, most of the problems raised by the use of C or FORTRAN as the programming interface can be elegantly avoided by using C++ as the implementation language.

4.2.2 Programming Interface in C++

C++ includes many programming features missing from the older HLLs, and these features can be used to hide much complexity from the actor programmer. In a C++ method programming interface, C++ **objects** representing each input path and each output path are passed to the method function as parameters. The class of each object depends on the data type carried by the datapath it represents. For instance an input variable read from a datapath which carries integers would be an object of a class `Mint` representing MeDaL-integers (integers with extra information and functions to facilitate the integer's passing through the MeDaL dataflow system). Objects of this class contain not only the integer's value, but the "stickiness" flag, and the run-time information about location within the dynamic "roadmap" (instantiated by the wrapper before the object is passed to the method function itself).

The MeDaL classes can contain not only information, but member functions which perform operations on them - such as retrieving the value from an input datapath, and placing a new value on an output datapath (with the sticky attribute set or unset). Essentially, these MeDaL classes form part of the method wrapper. However, by splitting the wrapper two ways (into the wrapper-function which is called by the run-time system, and the class code relating to input/output objects) it is possible to make the wrapper-function simpler - by making the MeDaL classes contain all possible generalised code, the wrapper-function need only contain any method-specific code. This in turn makes the function of the method-transforming module simpler.

The pre-defined MeDaL classes can include definitions of many commonly-used data structures, such as arrays, which can then be passed between actors through datapaths. However, using C++, programmers are not limited to the data types provided: they could easily implement new MeDaL data types by using **inheritance** to create new classes based on those provided. So long as a type is provided which can transfer blocks of memory through datapaths, any other more complex type can be built on top of this without needing technical knowledge of the interface with the run-time system. A minimal set of MeDaL classes is included in Appendix A.

A further way in which the C++ interface can hide technical detail from the programmer is in the provision of data-sending functions to be called during the actor method when data becomes ready for transmission on a datapath. While in C a function call (or a macro to hide one) must explicitly be used, this can be avoided in C++ by using the **operator overloading** mechanism. By overloading the assignment operator, a class member function can be called whenever a value is assigned to a MeDaL object; this function can check whether the object is an output object (corresponding to an output datapath), and if so, do whatever is necessary for it to be placed immediately on that datapath. None of this detail is visible to the actor method programmer, and naturally assignment to an output object can occur any number of times within an actor (for instance, within a loop). The assignment function provided with the class can also deal efficiently with copying; for instance, when an array from an input path is assigned to an output path, the assignment need only copy a pointer to where the array is stored in memory from one object to the

other. In other programming languages, this task would usually fall to the actor method programmer.

Other memory management issues can also be efficiently handled using C++; for instance, when an array from an input path is not copied to any output path, the memory taken up by that array should be "garbage-collected", and this can be done automatically by setting up a C++ **destructor** function for the class which frees any memory which is not marked as still being wanted (this marking being done by the assignment function). An example of this can be found in Appendix A in the MeDaL array (Mary) class.

Thus, by shifting much of the wrapper's complexity into standard class definitions, using the techniques described above, the remaining wrapper-function becomes very simple. Figure 4.2a illustrates typical wrapper code using the classes defined in Appendix A. The wrapper-function, `c0a2`, is what is actually called by the run-time system when it is found that all of this actor's input paths contain data. This wrapper then calls the real method function `actor2`.

```
void c0a2(unsigned company)    // company is passed in
{                               // by the run-time system
    ActorId me;
    ActorIdP meep = &me;
    Mint ip0, op0 ;           // actor's input and output
    void actor2(Mint&, Mint&);

    me.troupe = company;      // this structure holds the
    me.actor = 2;              // identity of this actor.

    ip0.setup(in, 0, meep);    // tell i/o objects whether
                                // they are input or output;
    op0.setup(out, 0, meep);   // WHICH input or output;
                                // and the actor's identity

    RTS_ActGoing(meep);        // tell run-time system
                                // actor has started

    actor2(ip0, op0);          // run the actor method

    RTS_ActOver(meep);         // tell run-time system
                                // actor has completed.
}
```

```

void actor2(Mint & in0,
            Mint & out0)    // pass by reference
{
    int inv;

    inv = 100 - in0;        // do something trivial
                           // (note type conversions
                           // are handled by C++)

    out0 = inv;             // assignment calls
                           // transmit function
}

```

Figure 4.2a: Wrapper and real method functions using C++ interface

A company must of course be passed to the wrapper at run-time because new companies can be created during execution. However, the other parts of the wrapper - the number of this actor within the company, and the numbers and types of the input and output paths - do not change. Actors with more than one input and output would simply have more lines of the form

```
ip1.setup(in, 1, meep)
```

which give each input/output object all the information necessary to call a generalised datapath receive or transmit function in the run-time system at the appropriate time - immediately, in the case of input objects (receives), and at assignment time in the case of output objects (transmits). The three arguments to the setup function are a token indicating whether the object is for input or output (actually an enumerated type); the number of that particular input or output, numbered from 0; and the pointer to the structure indicating what actor of which company is accessing that path.

The example above (and the code listings in the appendices) are taken from the shared-memory implementation of MeDaL; however, the type of architecture on which the MeDaL program is executing should not have a great impact on the functionality of the wrapper code, only the underlying run-time system. The run-time system, naturally, contains much architecture-specific detail. However, the general issues involved and techniques which can be used to implement them are discussed in the following sections.

4.3 Distributed-memory Run-time System

In comparing techniques used to implement the MeDaL run-time system on distributed-memory and shared-memory architectures, there are two important areas which account for most of the differences: the techniques used in transporting data items from one actor to the next, and the rôle played by the run-time system.

Both types of architectures have their strengths and weaknesses in both of the areas mentioned above. Dataflow is, in a sense, a message-passing paradigm; it is easy to draw comparisons between dataflow's actors and datapaths, and the processing nodes and links of message-passing architectures. However, this is not to say that a dataflow system of the type proposed here necessarily maps easily and efficiently onto a message-passing architecture. The problem is that since parallelism is extracted from a MeDaL program by the programming system rather than the programmer, the programming system must make the decisions about which nodes to place which actors on - moreover, it must sometimes do this at run-time. The load-balancing of dynamic processes is a non-trivial task, especially on a distributed architecture, on which load details of other nodes must be transmitted through the relatively slow inter-node links. The simplest strategy would be to have a single process on one node dedicated to doing this - but in a complex application, this process could become a bottleneck.

However, leaving such issues as load-balancing aside, before an efficient dataflow system can be produced, it is necessary to consider the basic techniques which can allow dataflow programs to run at all on distributed-memory multiprocessors. It was stated above that the two main issues are the implementation of datapaths between actors, and the rôle of the run-time system and the method code wrappers; these are dealt with in the following three sections.

4.3.1 Datapaths Between Distributed Actors

The basic mechanism for passing messages between processes on the iPSC/2 hypercube architecture, as with many other distributed multiprocessors, is explicitly to transmit messages, containing the data to be sent, to a specific destination. The operating system on the iPSC/2 (which runs on every node) provides functions to do this; on some other architectures, the application program itself would have to

handle routing of messages from one processor to the next until they reach their destination. Each processing node in the hypercube has a unique number, and since each node can run multiple processes, each process on a given node must be given a unique Process Identifier (PID) when it is started up. This message-passing mechanism can be adopted directly for the implementation of datapaths, the only caveat being that each actor must know the physical location of the actor to which each of its output paths leads. This is handled by the wrapper code, as will be shown below.

Conveniently, the iPSC/2's operating system provides not only a message-sending system call, but both blocking and non-blocking message-receiving calls; if data arrives for a process and no receive call is (blocked and) waiting for data items to arrive, the operating system places the data on a FIFO queue where it stays until the process requests more data. This provides everything necessary for the implementation of MeDaL's E-type paths. F-type paths simply remember the last item of data from the queue, and this datum is stored by the method-wrapper. The operating system detects and reports queue overflow which, as described in Chapter 3, is considered fatal to the execution of MeDaL programs; no "feedback" is needed from a consumer actor to its producer actor, to tell it whether or not the queue is full, so this does not constitute any message-passing overhead.

Naturally, data items which are sent to output paths using `send-sticky` are not actually sent to their destinations, but are rather stored by the method-wrapper to be passed back into the actor on its next firing. The optimisation of the dataflow graph (involving the removal of non-method actors and re-mapping of datapath destinations) described above also helps to reduce message-passing. All of these techniques by which the numbers of messages being transmitted can be reduced are important not only because message-passing is costly in terms of time, but because the bandwidth of inter-node communication links is limited, and too many messages can cause "congestion", leading to even greater delay.

It can be seen from the discussion above that the wrapper code around each actor's method code is of great importance in efficiently implementing dataflow programs on a distributed-memory architecture. The way in which the wrapper does its work, in co-operation with the run-time system, can now be described.

4.3.2 Run-time System for Distributed Actors

In a sense, the wrapper code around each actor forms part of the dataflow run-time system on a distributed architecture, handling the technical details of sending output data to its correct destination. To have execution of the dataflow program controlled by one central process would be a potential bottleneck both in terms of processing throughput and of inter-node congestion. However, as indicated above, there are situations in which processes need information about other processes on other nodes, and this will inevitably lead to inter-node traffic. These situations are as follows:

- for load balancing (even in the simplest possible terms, e.g. the initial allocation of new actor processes to nodes);
- for termination detection;
- for consulting the roadmap to find out the node and PID of the actor at the consumer end of an output path.

The latter situation occurs because of the need to keep the roadmap **coherent**. Since the roadmap can be extended dynamically at run-time, not all actors can be set up before execution begins; some will be "created" (i.e., their code copied to a processing node and set running) during execution, namely, when data is sent to them. In other words, the act of producing data for an actor which does not yet exist, causes its creation. Data could be produced for an actor *C*, before its creation, by two other actors *A* and *B*, each producing data for a different input path of *C*. If this happened simultaneously, and actor creation could happen in two places at once, two copies of the new actor *C* might be created - and neither might ever be able to execute, since it might never receive data on all of its input paths.

Thus, the ability to extend the roadmap must reside with one process on one node. There are two possible strategies for this, namely:

- The roadmap could be distributed between different nodes, and a key allowing extension could be held by any one node. A process wishing to acquire this key would have to request it from other nodes repeatedly until it was found and "given to" the searching process.
- One centralised copy of the roadmap could be held by a process always residing on the same node. This process would undertake any necessary extension of the map and creation of new actors.

Both of these strategies are viable, but each has its problems. The first has the potential for a higher number of messages, while the second has the potential to be a bottleneck. The first of these two approaches has been successfully adopted in implementing a virtual shared memory on the iPSC/2 [Lahj91], and indeed the handling of a dataflow roadmap could easily be implemented given a shared virtual memory. However, the development work described here pre-dates the completion of the virtual shared memory work which was in progress at the time. Ultimately, the second strategy was adopted for its far greater simplicity.

Thus, a system was adopted whereby a single process implementing a number of run-time system functions ran on a single processing node (the hypercube's host processor). This process, known as *Equity* (since all of the actor functions "belong" to it) maintained a single, global roadmap containing all of the information about which actors were located on which nodes (and with which PID). This roadmap was initially generated by the dataflow harness code incorporated into the run-time system at compile-time (see section 4.1.1).

The functions, therefore, of the *Equity* run-time system were to manage all the situations defined above in which inter-node co-operation is needed, in the following ways.

Load Balancing

Since *Equity* knows the physical location of all currently running actors, it has the only available information which can be used for load balancing. Thus, new actor processes can be placed on the nodes with the fewest already executing processes.

Termination Detection

Equity can co-operate with the actor wrappers (see below) and be informed whether an actor on a remote node is executing or not. In addition, any message sent to a halt actor anywhere in the dataflow graph is automatically routed to *Equity* so that it can perform the shutdown.

Creation of New Actors

It was stated earlier that the act of producing data for an actor which is not yet in existence causes the creation of the new actor. Essentially, when an actor's wrapper wishes to transmit data on a datapath for which it does not know the destination, it

sends a message to `Equity` requesting the appropriate information from the roadmap. If the actor does not already exist, `Equity` creates it before passing its location back to the actor which made the request.

In addition to these three functions, it is also `Equity`'s job to transmit data from source actors, and to handle program input/output (i.e., data to and from the input/output library actors). The former was necessary because source actors are virtually always too lightweight to be worth the cost of distributing to a remote node, and the latter because all I/O on the iPSC/2 must go through the host processor anyway. This being the case, it was simplest for all messages to output actors to be sent automatically to `Equity` to for processing.

4.3.3 The Wrappers of Distributed Actors

As explained above, the rôle of the actor method wrapper is to handle technical aspects of dataflow execution from the actor programmer. Obviously, because actors can only communicate with other parts of the program through datapaths, the implementation of receiving and sending data through datapaths form the main basis of the wrapper code.

Considering transmission first, it was described above that when an actor wishes to send a data item to an actor the location of which it does not know, it requests the location from `Equity`. In fact, initially, the actor does not know the location of any of the actors to which its output paths go; a form of **late binding** of output paths to their destinations is employed. What happens is as follows: when an actor wishes to place data on an output path, it calls a function which is part of its wrapper (since C++ was not available on the iPSC/2, this was by means of an explicit function call). The parameters to this call are the data item to be sent, and the unique number of the datapath on which it is to be sent.

This wrapper maintains a **lookup table** mapping each output datapath to the node, PID, and input port number of the actor which form the destination of that datapath. If the entry corresponding to an output path is empty, the wrapper sends a message to `Equity` requesting the destination to that output path (identifying the path by specifying the current actor's company identification, number within the company, and the number of the output path). The wrapper code then blocks until `Equity` sends back the node, PID and input number of the input path and

destination actor, and when this arrives, places this information into the lookup table. The important point is that this lookup table is persistent with relation to the actor; so once the information has been found, it will not cause message-passing to find it again for subsequent transmits on that datapath, even during future executions. This approach reduces the amount of message-passing needed, and hence speeds up execution; though it does make subsequent load-balancing difficult.

In fact, the run-time system can be completely passive under this arrangement; all communication is initiated by the actors themselves (or their wrappers), and the run-time system simply responds. Since the wrappers need not deal with messages from the run-time system except when one has specifically been requested, this makes them reasonably simple.

It was mentioned above that actor wrappers maintain a datapath destination look-up table which is persistent relative to actor firings. This is possible because the processes in which actors and their wrappers run are not destroyed once the actor has terminated. Since MeDaL graphs can contain cycles, and MeDaL actors can place more than one item of data on an output path during one execution, it is not possible - in the general case - to know whether, once an actor has completed its work, whether it will be able to fire again in the future.

Therefore, the following procedure must be followed. Once an actor completes, and control returns to the wrapper, the wrapper checks the operating system's input queues for its process, and if all are non-empty, runs again immediately. It checks these queues using the asynchronous checking call, `iprobe()` which does not block until there is data in the queue. However, if any of the queues are empty, it sends a message to `Equity` stating that its actor is no longer working, and blocks until data arrives on that queue (using the synchronous call, `cprobe()` so that its process is suspended, freeing that processing node). When data is received, such that all paths again contain data, the wrapper sends a message telling `Equity` that its actor is working once more.

In this way `Equity` can detect termination, which might arise through all actors completing their work, but can also happen if deadlock occurs. In many programs there will be far more of these messages than any other type between the nodes and the run-time system (in the iterative programs used for experimentation

they made up between 84%-90% of the messages received by `Equity`); however, since the messages are short, can easily be processed by the run-time system, and can be sent asynchronously (the actor does not need to wait for a reply), this does not significantly slow down the execution. There are, however, more sophisticated algorithms which could be adapted, a number of which are discussed in [Matt87].

This section has described the issues involved in implementing a run-time system which supports all the main features of `MeDaL` on a distributed-memory architecture such as the `iPSC/2`, and what techniques can be used to address these issues. However, the work described here does not go beyond demonstrating that such an implementation is possible. Because of the lack of experimental time available on such an architecture, no claims can be made that such a system would provide a viable system for implementing parallel programs, in terms of time-efficiency compared to other systems.

4.4 Shared-memory Run-time System

Even though dataflow can be seen as a message-passing model, the availability of shared memory makes implementing dataflow programs (and, arguably, parallel programs in general) easier than it is on a distributed architecture in a number of ways. In particular, there is no need for mechanisms to work out which processor a particular actor is running on, and since the run-time system code is easily accessible to all actor methods, the actors themselves can execute this code (via run-time library function calls) rather than requiring a separate process to do this.

The `Encore Multimax`, like other symmetric shared-memory multiprocessors, provides HLL-level support for parallelism in the form of a library of parallelism primitives for creating parallel processes and for mutual exclusion. The parallel process mechanism involves the creation of "threads," which are essentially "lightweight" processes in the sense that they are not costly in terms of time and space to create and destroy. If the program creates more threads than there are processors available to run them, the system handles scheduling of the threads through its own run-queue; the programmer's view is that all threads run concurrently. This system is convenient because, as it has often been noted, most

applications contain more potential parallelism than is available on current MIMD multiprocessors [Vali90]. Thus, if more processes are set going than there are processors, this ensures that all processors will be kept busy all the time, ensuring near-optimum efficiency (subject, of course, to the overhead of process creation and destruction).

As described earlier in this chapter, the run-time system implemented on the shared-memory machine provides a generalised (not tailored) set of functions; in fact, it provides a second run-time library, which forms an interface between the method wrapper code and the threads library.

The shared-memory run-time system can, like the distributed-memory one, be considered as the sum of three separate parts: the code which implements the datapaths, the method wrapper code, and the rest of the run-time library. The following sections deal with each of these in turn.

4.4.1 Shared-memory Datapaths

Datapaths can be easily implemented using shared memory, which is therefore accessible to both of the actors which use it, namely the producer and the consumer. In fact, using the merge-actor optimisation described in section 4.1.3, there may actually be several producers writing to the tail of one datapath queue. Mutual exclusion must therefore be used to ensure consistency of the datapath structure, and this can be easily implemented by creating a lock associated with each datapath, which must be acquired before writing can take place (note that the consumer also alters the queue by removing the head element, and so must also acquire the lock). This lock can be a simple parallelism primitive such as a semaphore.

A simple bounded-buffer algorithm can be used to implement the FIFO queue needed for datapaths. Since MeDaL datapaths are theoretically infinite, it is necessary for the producer to extend the buffer (assuming memory is available) when it becomes full. This results in the producer holding the path's lock for longer than normal. However, simple heuristics can be added to determine by how much the buffer is extended each time it becomes full, to ensure that this happens relatively infrequently; and in any case, memory allocation is generally not a heavyweight operation.

4.4.2 Shared-memory Wrappers

The method code wrappers needed in a shared-memory implementation are almost identical to those in the distributed-memory version; their main purpose is to set up the C++ input/output objects with all the information needed to receive and transmit data through datapaths - information which is not known until run-time. The classes for these input and output objects contain member functions (such as the overloaded assignment operator) which use this information to call functions in the run-time library. Since these are simple function calls, the run-time library code is executed by the actors themselves - even when this involves extending the roadmap (sections of the roadmap also being protected by locks, to prevent two actors altering a section concurrently). So, although the code for roadmap maintenance is part of the run-time system, and its details do not need to be visible to the wrapper or actor method levels, this code is executed in concurrent processes, achieving greater parallelism than the simple centralised-roadmap scheme adopted on the hypercube machine.

The shared-memory run-time system itself is not included in this thesis since it is highly machine-specific; however, its function call interface is clearly illustrated in the MeDaL classes listed in Appendix A. All function names beginning `RTS_` are calls to run-time library functions.

Because a generalised run-time library is used, an extra wrapper layer is needed to provide the dataflow harness describing the layout of a particular dataflow graph. This takes the form of an extra wrapper function, one for each company in the MeDaL program. Each company harness function is called in turn by a single further function, which is effectively a wrapper for the whole program, in the following way.

When starting execution, control enters the run-time library (not the programmer's method code), which calls the whole-program wrapper function. Since the run-time library is pre-compiled, this function must have a specific name such as `RegisterCompanies()`. This function embodies knowledge of the company harness functions and calls each of them in turn, allowing them to **register** themselves with the run-time system: they call run-time library functions which place appropriate numbers of actors and datapaths in a data structure (the company register) along with pointers to the actor wrapper functions, the destinations of the

datapaths and so on. Thus, after every company harness function has been called, the run-time system has a record of the layout of each company.

The company data structure used in the register of companies, however, only contains static data about the internal layout of each company. This data structure is then used during execution as a template, from which the dynamic roadmap can be built. The roadmap itself is a set of dynamic structures called troupes, each of which is essentially a company, with the addition of dynamic data such as the physical locations of datapath memory buffers, locks to protect them, etc. When data flows into a troupe for the first time during execution, the troupe is instantiated with the appropriate actors and datapaths, using the template in the company register. In this way, the roadmap can be dynamically extended very quickly (and, indeed, potentially garbage-collected) without altering the information needed to create further troupes.

Since troupe creation involves the creation of data structures for an entire company at once, it is of course a relatively heavyweight operation compared to the placing of data on a datapath; however, it occurs correspondingly more rarely - in fact MeDaL programs can be developed without using troupes at all. Moreover, it occurs within the threads running actor processes, and so two different new troupes can be set up concurrently.

Both datapath extension and troupe creation occur as a result of an actor method transmitting a data item on a datapath. In fact, time spent within the transmit function accounts for almost all the time the actor spends executing run-time library code. Since acquiring a datapath lock (or, relatively rarely, a roadmap lock in order to extend the roadmap) is the only action which may force actors to synchronise with each other, it is important to consider the algorithm for transmitting data items on a datapath, in order to find the minimum time for which these locks must be held. Figure 4.4a is a simplified, pseudo-code version of the algorithm used. Essentially each datapath has a lock; each actor has a lock; the whole roadmap has a lock; and a global count of running actors has a lock.


```

acquire path_lock;
place data at tail of datapath;
if (path is now full) {
    extend path; }
release path_lock;

acquire roadmap_lock;
if (destination is in a new troupe) {
    create troupe from appropriate company;
    add troupe to roadmap; }
release roadmap_lock;

acquire destination_actor_lock;
if ((actor is a depth actor)
    or (actor is not running)) {
    enabled = True;
    for (index = 1 to number_of_input_paths) {
        if (input_path[index] is empty)
            enabled = False; }
    }
    if (enabled) {
        mark actor as running;
        acquire running_lock;
        increment running_count;
        release running_lock;
        create new thread to run actor; }
    else {
        release actor_lock; }
else {
    release actor_lock }

```

Figure 4.4a: Shared-memory transmit algorithm

Note that when datapaths are created, they are allocated a certain amount of empty space to start with, and they are extended whenever they become full, so that the next item of data can quickly be added to the tail of the path.

The reason why the roadmap must be locked before even testing that the destination is an (as yet) uninstantiated troupe, is that two different datapaths, being written-to concurrently, may both have destinations in the same uninstantiated troupe; so a situation in which both of these try to instantiate the new troupe must be avoided.

The destination actor must also have a lock which is acquired before the sender begins to check whether the actor is enabled for execution, to prevent another transmitter to the same destination actor from firing it, thus removing items from the queues, during the process. However, testing each of the actor's input paths for

non-emptiness does not involve acquiring the lock for each datapath, since adding further items to a non-empty path will not affect enablement. It can be seen that, if the destination actor is enabled, its `actor_lock` is not released by this algorithm. The lock is in fact released by the `RTS_ActGoing()` function in the actor wrapper (see Figure 4.2a). This is called after data has been read from all of the actor's input datapaths. Another actor adding data to the input paths of the actor can then enter the critical section and check whether to fire another copy of the actor.

The algorithm above was chosen on two criteria: firstly, to be as simple as possible, and secondly, to ensure that actors are fired as early as possible. The second criterion relies on the underlying parallelism package to place new actor threads on a run-queue if no processors are currently available to execute them. A more sophisticated algorithm could be used to ensure that there were always as many actors executing as processors available, to avoid the thread creation and termination overheads by starting a set number of threads which would execute one actor and then another. However, such an algorithm would also need to maintain a queue of enabled actors, in order to avoid having to check the enabled state of every actor in the roadmap. Assuming the overhead of the two queueing mechanisms was comparable, the only difference then would be the thread creation overhead; and since the point of threads is that they are lightweight to create and destroy, this possible gain was not deemed to be worth the extra complexity of code.

4.4.3 Other Functions

Apart from creating the company register, maintaining the roadmap, handling the transmitting and receiving of data items through datapaths, and firing actors, the run-time library's other main task is to support the MeDaL library actors. Since some file handling operations can take orders of magnitude longer than the operations done within the transmit function described above, file handling is not done within actor threads. Instead, a dedicated input/output handler thread must be set up, to monitor any input files (including the standard input) and accept any data from them, using the normal transmit function to place it on a datapath; and to accept any data items sent to file outputs (including the standard output) and write them to the appropriate file. Data items sent to file output actors are handled in a special way in the transmit function (but not illustrated above, for simplicity);

rather than being placed in a datapath like other items, data to be sent to a file output actor is placed in a specially designated buffer of shared memory. The run-time system's input/output thread monitors this buffer, so as to retrieve new output data to be sent to a file.

The halt actor is very simple to implement using shared memory. This, too, is handled in a special way by the transmit function; when data is sent to a halt actor, a global flag is simply set indicating that no actors may be fired. This flag is checked in the transmit routine (again, not shown above for simplicity) at the same time as the other conditions for firing (whether the destination actor is a depth actor, or is not already running). Thus, after the flag has been set, all currently executing actors run to termination as normal, then shut down gracefully, but no new actors are started.

The final function of the run-time system is to detect termination, and this too is very simple, using the `running_count` shown in figure 4.4a. When this count reaches zero - whether through normal circumstances or through the halt flag preventing any enabled actors from firing - termination of the program occurs.

Using the techniques described in this section, a fully working shared-memory run-time system has been implemented on the Encore Multimax, and the performance of this run-time system, and of the way in which it executes programs, are described in the next chapter.

4.5 Summary

This chapter has described the issues and problems involved in implementing a programming system based on the MeDaL notation, on both shared-memory and distributed-memory multiprocessors, and has presented techniques and solutions to address them.

Although MeDaL was designed for use in the whole software life-cycle of design, implementation, debugging, performance tuning, and maintenance, the work described in this chapter concentrates heavily on software support for implementation, though clearly the design of the programming system has some implications for design and debugging. However, success in implementing MeDaL programs is the key to the success of the whole MeDaL concept, since unless

efficient implementation is possible, MeDaL cannot be considered a successful parallel programming tool.

This chapter has sought to demonstrate that it is possible to implement MeDaL programs directly on both main types of MIMD architecture. Indeed, the implementation techniques described have proven successful, in the sense that the major parts of the programming system described here have been implemented and real MeDaL programs have been run, on both architectures used. However, this is of course a very limited definition of success. In order to consider the fuller picture, the efficiency and effectiveness of software developed using MeDaL must be examined, and this forms the focus of the next chapter.

Performance Evaluation

Having demonstrated that it is possible to implement a programming system based on MeDaL on medium-grained multiprocessors, a far more complex question arises: one of whether such a system is worthwhile. The question is complex because there is no general way in which one can completely evaluate the efficiency and effectiveness of a programming system. This complexity arises from the general-purpose nature of programming languages and systems; they can be used for many different purposes, and the criteria for success varies between different cases.

Because MeDaL is designed to support all the phases of the software life-cycle, there are many components which must be designed and tested in order to obtain information about MeDaL's suitability for those phases of software development (these modules and their rôles were described in Chapter 4). The implementation and evaluation of all of these modules is beyond the scope of this thesis. However, it can be seen that the MeDaL run-time system is a key component on which the success of the whole system hinges. MeDaL actor code does not contain parallel constructs: it is the run-time system which manages parallelism for the whole program. It is therefore vital to the success of MeDaL as a *parallel* programming system, that this management of parallelism by the run-time system can be implemented efficiently enough for programs implemented using MeDaL to exhibit speedups when running on multiple processors. If this is not so, then clearly the exercise as a whole can be regarded as not worthwhile.

The term "speedups" used above is a vague and somewhat controversial term, since it implies a comparison, but it is not clear exactly what the comparison should be between. Should the time taken to execute a program in parallel be compared with: the same program running on only one processor; a program with the same functionality written purely sequentially, without any parallel constructs; a program with the same functionality using an alternative method of parallelisation; or a theoretical "ideal" version of the algorithm being used? None of these is an entirely

fair comparison. However, as the primary aim of this chapter is to examine the behaviour of the MeDaL system, the latter two can reasonably be ignored. Comparison to alternative methods of parallelisation is of only limited interest, since it would provide as much information about the other parallelisation methods as about MeDaL. Comparison to ideal versions of the algorithm being parallelised would provide information about the algorithm rather than about the performance of the MeDaL system. Thus, this chapter is limited to a definition of speedup whereby

$$\text{speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

Where T_{serial} is the time taken by a serial version of the program - either a MeDaL program running on one processor (hence serially) or a program based on the same algorithm but without using MeDaL or any other parallel constructs.

Using this definition of speedup, when employing 2 processes (in this case, actors) to do some computation, the ideal would be that the speedup would be 2 (i.e. that the time taken would be half as long). When employing 3 processes, the ideal speedup would be 3, and so on - this is known as linear speedup. In practice, however, as stated in Amdahl's Law [Amda67] there is usually some part of the program, involved with starting up parallelism, which cannot itself be parallelised - it must remain serial. Thus, there is an upper bound on speedup which means that linear speedup can rarely, in practice, be achieved.

An alternative to measuring speedups is to measure the efficiency of a parallel program. One useful measure of a parallel program's efficiency is the proportion of the program's total execution time which is spent executing that code which actually performs the "useful" algorithm, rather than the code which implements the parallel constructs. Of course, the structure of the parallel program has a significant effect on efficiency - an algorithm which requires frequent synchronisation between processors, for instance, is likely to be inefficient. Moreover, a programming language can be used to express a wide range of programs (usually each having a number of possible structural forms). Thus, it is not possible to prove *in general* that MeDaL programs are efficient.

However, it remains that an efficient implementation is needed to underpin an efficient design of any parallel program; so if it can be shown that the means of

parallelisation used by MeDaL do not, in themselves, make parallel programs unacceptably inefficient, it follows that efficient parallel programs can be implemented in MeDaL. Therefore, this chapter concentrates on evaluating the efficiency of an implementation of a MeDaL run-time system which provides the parallelism for MeDaL programs. Only once this has been covered are the speedups recorded on an example application presented, to provide a general guide to the success of MeDaL. The remainder of the chapter describes the implementation environment, and the key parts of the implementation which were measured. The experimental results themselves are then presented and evaluated.

5.1 Implementation Environment

Since the work implementing a MeDaL run-time system on a distributed architecture (the Intel iPSC/2 - see Chapter 4) was of too limited a time period to yield a fully working implementation, this chapter deals only with the implementation on a shared-memory multiprocessor, the Encore Multimax. The hardware used was a Multimax 520 system, with 14 30MHz NS32532 processors and 96MBytes of memory. The processors in Multimax systems are fully symmetric, and communicate with a logically single shared memory, through a common bus with a typical transfer rate of 100MBytes/second. There are two processors to each processor card, which share 256KBytes of write-deferred cache memory on each card. The mention of cache memory is relevant because its presence can have a significant effect on the observed execution times of parallel programs, not only because access to cache memory is faster than access to main memory, but because cache hits reduce the need for access to memory through the bus, thus reducing the chances of contention between processors for the bus. This latter effect can be a major cause of loss of efficiency in shared-memory parallel processing. The maintenance of cache memory in shared-memory multiprocessors is relatively complex, since **cache coherency** (consistency between two or more caches which may hold the same word of shared memory) must be maintained. The mechanisms for providing cache coherency, however, are transparent to the application programmer.

The operating system available on the Multimax was UMAX4.3, a variety of Unix similar to BSD4.3, though with System V-like extensions (including the

provision of shared segments) and system calls to manage concurrent access to shared memory. Parallel programming is possible at the operating system level, since Unix processes are scheduled to run on any available processor. However, the synchronisation mechanisms available at this level are *heavyweight*; they require many instructions, and take of the order of milliseconds to execute. To be efficient, therefore - by reducing this overhead to an insignificant proportion of the total execution time - the size of the computations undertaken by each processor would have to be correspondingly large, certainly of the order of thousands of instructions. Thus, concurrency at this level could be described as large-grained parallelism.

However, parallel programming at a finer-grained level is also provided for on the Multimax, in the form of run-time libraries which provide synchronisation functions which operate directly in shared memory without using the concurrency system calls provided by the operating system kernel. This effectively gives the programmer a way to bypass heavyweight operating system constructs. The finest-grain interface available is called EPT, the Encore Parallel Threads library - a **thread** being the lightweight equivalent of a Unix process. Using this library, programs can be built in which the individual processing tasks can run as (potentially concurrent) threads; these threads can be created and destroyed with a much smaller overhead than that of the creation and destruction of Unix processes, enabling programs to be efficient at a finer grain size.

In fact, threads run within a number of Unix processes, which are created at the start of the program's execution and not destroyed until the EPT-based program terminates. In other words, there is a hierarchy:

- **threads** (medium-grained) can be run in any of a group of participating
- **unix processes** (large-grained) which can execute on any available
- **processor**.

The EPT run-time library handles the decisions about the scheduling of threads within the available processes, and the operating system handles the decisions about the scheduling of processes on processor hardware. Thus, application programmers only need to structure their programs into individual threads, and handle the synchronisation between these threads using primitives such as semaphores provided by the EPT library. These synchronisation primitives must

also be used to regulate access to shared memory; the onus of shared memory management is on the programmer.

This is the programming environment in which many parallel applications are developed on the Multimax (and similar environments are available on similar architectures). However, the management of shared data structures and synchronisation between threads can require much programmer effort. MeDaL aims to provide a "higher-level" programming system in which *all* control of parallelism is handled by the system, and the programmer need only write sequential processes.

5.2 MeDaL Run-time System Implementation

The MeDaL run-time system undertakes the tasks of memory management and the management of, and synchronisation between, medium-grained processes (threads on the Multimax). It does this by providing a second run-time system "between" the MeDaL programmer's sequential code sections and the EPT library. Since MeDaL uses a dataflow model, process management is via the firing of actors (each actor being executed in one thread), and the only access to shared memory (requiring synchronisation to prevent concurrent access) is through datapaths; so these two functions (actor firing and datapath management) are the main tasks which the MeDaL run-time system performs. Sections 5.3 and 5.4 evaluate the efficiency of the implemented run-time system, but first it is necessary to understand specifically what the run-time system does. This section briefly describes the implementation of the MeDaL run-time system in the environment described in the previous section.

5.2.1 General Configuration

As discussed in sections 4.1 and 4.4, the approach taken on the shared-memory architecture was to provide a generalised run-time system. Thus, the final runnable parallel program consists of:

- the actor code, written by the application programmer;
- the actor wrappers, which envelop each actor in an EPT thread;
- roadmap-generating code, derived from the MeDaL graph;
- the generalised MeDaL run-time library, which makes calls to the EPT library.

The first three components are compiled from the MeDaL graph and actor code supplied by the programmer; these are then linked to the MeDaL run-time system library (called RTS) and the EPT run-time library (`libept`). Control enters the program through RTS, which sets up the thread environment via calls to `libept`. The RTS then calls the roadmap-generating code, which in turn calls RTS functions to **register** each company (consisting of actors and datapaths) into the roadmap maintained by RTS. In this way, the functions of the RTS are generalised - the code for each specific MeDaL application is separate.

Once this roadmap-generating code has finished executing, the RTS runs the source actor code for the root company; the data items placed on datapaths by the source actors normally cause other actors to fire, and so execution of the MeDaL program, potentially in parallel, begins. The procedure described above is essentially a "set-up" procedure necessary before parallel execution can begin, and as such imposes an overhead on the execution time of the program. However, the extra time needed to perform this (over a program written directly in the EPT environment without using MeDaL) only amounts to a few hundred instructions, or a few thousand for a very complex application. Given that parallel programming is only needed in cases where large amounts of processing is done (millions of instructions at the least), this set-up overhead is not considered significant, and will not be considered in detail.

5.2.2 Specific Functions

The reason the set-up overhead is insignificant is because it is incurred only once for each execution run of the program, and is therefore an overhead of the whole program. However, MeDaL memory management (e.g. the maintenance of datapaths) and actor-firing functions are called repeatedly during the lifetime of the program, and therefore have a much more important effect on the overall efficiency of the program. These operations are now described in more detail. The main functions are:

- Transmit an item of data to an output datapath;
- Receive an item of data from an input datapath; and
- Yank a "sticky" item of data back from an output datapath.

The algorithms for each of these are discussed in turn below. However, there is one common factor: each of these functions alters the contents of a datapath. To prevent

concurrent access, which could potentially leave the datapath in an inconsistent state, each datapath has a lock associated with it (in fact, a semaphore) which must be acquired before any operation can take place. Consider the following scenarios:

- Two actors share an output datapath (where a merge actor has been optimised out). Since any two actors can potentially run concurrently, they could both attempt to write at the same moment.
- An actor *Z* has two input paths *A* and *B*. *A* contains several items of data, *B* none. While one actor writes to *A*, another writes to *B*. The latter causes *Z* to fire, and so read data from both *A* and *B*. Therefore, there could be an attempt to both write to, and read from, *A* at the same time.

These scenarios illustrate that, in general, both read-write and write-write conflicts can occur on datapaths. A simple scheme was adopted by which the datapath's lock must be acquired before any read or write operation proceeds. A more sophisticated scheme might, for instance, check for the existence of other actors sharing a datapath, or for other inputs to the destination; however, the extra complexity of code needed to do this was not felt sufficient to justify its inclusion in this simple implementation. With the principle in mind that the datapath's lock must be acquired before any operation on it, the three main functions *Transmit*, *Receive* and *Yank* may be described.

When an actor wishes to transmit an item of data on its output port, it simply passes the data (or, in the case of structured data such as arrays, a pointer to the data) and the number of the output port (output ports being numbered, left to right) to the RTS function *Transmit*. This function consults the roadmap to map the output port to a datapath and then acquires the lock for that datapath, possibly blocking if another actor already owns the lock until it is released. Having done this, the next step is to place the data item onto the tail of the datapath's queue structure, extending the size of the queue if necessary. This size extension requires the allocation of a new block of memory for the queue, and the copying across of the elements from the old to the new memory areas - a relatively heavyweight operation. To ensure that re-allocation does not happen often, whenever the queue becomes full its size is doubled; therefore, datapath extension rarely causes other actors to block (more complex algorithms for extension involving heuristics, or non-contiguous memory areas, would also be possible). Having added the data item to the queue and possibly adjusted its size, the datapath's lock is released.

Transmitting a data item may, of course, result in the datapath's destination actor becoming enabled, and so firing; this too takes place within the `Transmit` function. Having added the data item being transmitted to a datapath, `Transmit` looks up the path's destination actor. If the actor is known to be a general-purpose (non-depth) actor which is already running, it cannot be fired immediately, and so no further action is taken. Otherwise, `Transmit` now acquires a lock pertaining to the destination actor (to prevent two actors checking for fireability concurrently), then the lock for each of that actor's input paths, checking as it goes along that each path is non-empty; if a path is empty, the procedure is aborted and all locks already obtained are released. If, however, all paths are found to be non-empty, the actor is fired, by telling EPT to run the actor's wrapper as a thread. EPT places the new thread on its run queue before returning (the new thread runs immediately only if a processor is available). This EPT operation adds a significant overhead to the time taken by `Transmit` (see below), but only in the case where `Transmit` does actually result in the firing of another actor. Finally, the datapath locks are released again. Note that if an actor is fired, `Transmit` does not release the destination actor's lock - see `Receive`, below.

It can be seen from the discussion above that `Transmit` acquires and releases the lock associated with its datapath, and then acquires and releases a number of further locks, depending on how many input paths the receiving actor has, and what these input paths contain. At best only one path lock will be acquired and released; at worst, for an actor with n inputs, there will be n acquisitions and n releases (in the case where only the last path is empty).

Turning to the `Receive` function, this is called by the actor wrapper code (when it is run as a thread by EPT). `Receive` is called for each input path. Like `Transmit`, the number of the input path must be mapped to an actual datapath (paths being numbered from left to right as usual). The datapath's lock is acquired, the data item at the head of the queue is read, and the queue size decremented; then the lock is released. No further work need be done by the `Receive` function. Once `Receive` has been called for each of an actor's input paths, the actor wrapper can release the actor's lock, allowing any `Transmits` in progress to check whether it can be run again.

The `Yank` function is similar to `Receive`, except that it receives data from a

given output path, rather than an input path. Like `Receive`, it is called by the actor wrapper as part of the firing procedure; the actor wrapper checks each output path in turn to see whether a "sticky" item of data is present. Since only the producer can affect sticky data items, there is no need to acquire the datapath locks for this unless the producer is a deep actor; though the mapping from output path to physical datapath must be done. When sticky data is detected, `Yank` is called on that datapath. `Yank` simply acquires the datapath lock, reads the tail item of data, decrements the queue size and releases the lock. Thus, the data item is completely removed from that path; it is up to the actor to re-transmit it (stickily or otherwise). Since the mapping of output path to datapath has already been done, `Yank` appears to take slightly less time than `Receive` (see below). However, the two operations are in fact directly equivalent.

Having discussed the functionality of these three key components of the RTS, and shown what the algorithms involve, the actual times taken to execute these algorithms can now be examined.

5.3 Experimental Results

To determine how efficient or inefficient the presence of the MeDaL RTS makes a parallel program, one first needs to know the amount of execution time spent in the RTS as a fraction of the total execution time of the program. Therefore, the starting point to measuring this fraction is to measure the time taken by the three main RTS operations, `Transmit`, `Receive` and `Yank`.

The Multimax used for this experimentation is a multi-user timesharing system running a complex operating system. As it was not possible to obtain sole use of the machine for experimentation, all programs were potentially subject to interference from other processes running on the machine at the same time; while this interference does not affect the functionality of programs, it does affect the timing, since at any time the program may be suspended to allow other processes the use of processors. Therefore, in order to approximate a true picture of the time taken for particular operations, the test programs were each run approximately 30 times, and the *lowest observed time* was recorded. The lowest observed time represents the program run in which there was the least operating system interference, and so provides the most accurate approximation to the real cost (in

terms of time) of the RTS operations. In addition, all experiments were run at times when the average system load was less than 1.0 (i.e. there was on average less than 1 runnable user process per minute, across all 14 processors).

Of course, if the test programs had been run more times, lower execution times might have been recorded; however, the standard deviation of results was found to be sufficiently small not to warrant more runs. The mean execution time was typically 10% higher than the lowest observed time. It should also be noted that all of the test programs were run in the same execution environment, namely within an EPT thread. This is true even in those cases below which do not involve parallelism, to enable fair comparisons.

The following sections give the results of experimentation first with the basic RTS functions, and then with more complex programs which incorporate them. The performance of a real application program - the matrix multiplication example from Chapter 3 - is then presented and discussed. Finally, the efficiency of these programs is examined.

5.3.1 Basic Functions

Figure 5.3a shows the time cost of the main RTS functions. The timings, of course, relate to the hardware platform used and are of no significance to other (perhaps newer) hardware; for this reason, the number of source-level instructions involved is also given. However, these too are of only limited value since some of the instructions are in fact function calls to the EPT library, which may then in turn call the operating system; so all "instructions" are not of the same cost. To further aid comparison, the times taken to do tight loops of 100 floating-point multiplications, and 100 integer multiplications, are also given.

Operation	Instructions	Time (μs)
Yank	13	35
Receive	16	37
Transmit	64	118
Transmit + fire	74	445
EPT thread startup	N/A	161
100 flop mults	100	277
100 int mults	100	146

Figure 5.3a: Basic RTS functions

It can be seen that the call to `libept` to start up a new thread (or more accurately, place it on a run queue) forms a significant part of the cost of any `Transmit` call which involves firing another actor: 36% in the case of firing a two-input actor. In fact, it was observed that the first few calls to EPT to start up a new thread took significantly longer than subsequent calls; this is thought to be due to memory allocation overheads and a cacheing effect (the relevant code is present in cache after the first call, and so can be executed much more quickly). The cacheing effect was also observed in other functions, so the minimum observed times are from programs in which the appropriate functions were called several times before the timing was recorded, to ensure presence in the cache.

There are a two other minor points to note about the timings given in Figure 5.3a. Firstly, the timings given for `Transmit` refer to a specimen situation in which the destination actor has two inputs. Timings for `Transmit` to actors with fewer or more input paths were found to vary by only a few microseconds; this could extend into the tens of microseconds only for actors with large numbers of input paths, a rare occurrence. Secondly, for comparison with the timings given for floating-point and integer multiplications, it was found that in 500 microseconds, 182 floating-point multiplications or 327 integer multiplications could be executed. These figures will be referred to later in this chapter. For floating-point multiplications, this is equivalent to around 0.5 megaflops, a performance figure which is verified by benchmarks such as LINPACK. The integer operation was deliberately chosen to be a poor case, and equates to around 1 MIPS; benchmarks show that for other integer-based operations, the Multimax can reach 7.5 MIPS.

Having determined the time-cost of the basic RTS functions, the overhead caused by their use in parallel programs can now be considered.

5.3.2 Use of Basic Functions within Programs

Although Figure 5.3a above gives an indication of the minimum overhead of using the MeDaL RTS functions, these overheads do not constitute the whole cost of using RTS for the parallelism constructs and shared-memory management of parallel programs. In addition to these costs, there are the cost of the actor wrappers and the cost of the programming language interface between RTS and whatever

language the actors are written in; in the case of this implementation, C++. Whenever an actor wishes to transmit an item of data on a datapath, an overhead is incurred before the RTS is even called. In the case of C++, transmission is triggered by an assignment to a variable representing an output path; assignment is overloaded, so a function which handles assignment is called in the C++ library (essentially part of the actor wrapper) which in turn calls the RTS.

These overheads are important because it is their cost which will determine how efficient parallel programs which use the MeDaL RTS can be. In particular, if the overhead of the RTS functions grows as the amount of processing being done by the rest of the program grows, this would place an upper bound on the range of algorithms which can be efficiently implemented using MeDaL. Therefore, this section examines whether this is in fact the case.

Rather than trying to time each such overhead individually, it is more informative to look at the overall picture using program scenarios. To examine these scenarios, consider a parallel program which consists of two processes, A and B, which require time T_a and T_b to complete, respectively. Clearly, the fastest possible sequential version of this program would take time $T_a + T_b$ to run, and the fastest possible parallel version (in theory) would take $\max(T_a, T_b)$. Assume further that $T_a = T_b$, then in theory a parallel version (with no data dependency between the two processes) could execute both processes in time T_a . The question now becomes, what overhead does the use of the MeDaL RTS in practice add to this time? This depends on exactly what the relationship between A and B is. The possible scenarios are:

1. A and B are the same code, operating on different input data.
2. A and B are two different pieces of code, operating on the same or different input data.
3. A and B are different pieces of code, and one is dependent on the other.

In MeDaL, scenario 1 is represented by a depth actor; two items of data are sent on its input path and two copies of the actor are fired. Scenario 2 corresponds to horizontal parallelism, where the two data items are transmitted on two separate datapaths, causing each of the two actors to fire. Scenario 3 corresponds to vertical parallelism, in which the output path of A, for instance, is the input path of B.

To look at these scenarios another way, in scenario 1, actors A and B share two

datapaths - the input path and the output path of the depth actor. In scenario 2, no paths are shared - A and B have separate input and output paths. In scenario 3, one path, the path between A and B, is shared. The presence of shared datapaths is important because, as described earlier, access to datapaths must be mutually exclusive - thus when two actors both require access at the same time, an unavoidable extra overhead on execution is created.

Scenarios 1 and 2 are the most interesting because they constitute the worst and best case, respectively, when considering the use of RTS functions. In the case of the depth actor, both actors must contend for access to a datapath both when reading their input data and writing their output data. In scenario 2, neither actor has to contend when either reading or writing.

Figure 5.3b illustrates what happens during execution of scenario 1. There are assumed to be three processes running concurrently: one which transmits the initial data, and two which receive the data, process it, and transmit further data. Each of these processes consist of several individual activities, which are labelled as follows. The label T represents the Transmit function; the label R the Receive function; A the actor process; and T', a transmit function which does not result in an actor firing, since any further actor firing can be considered the overhead of that actor, rather than actor A. The times taken by these activities are T_t , T_r , T_a and T_t' , respectively. Dotted lines in the diagram indicate the presence of dependencies between the processes; the order of execution illustrated has both Transmit functions occurring before either Receive is allowed access to the datapath which they all access. This need not, in practice, be the case; however, the overall time is not affected.

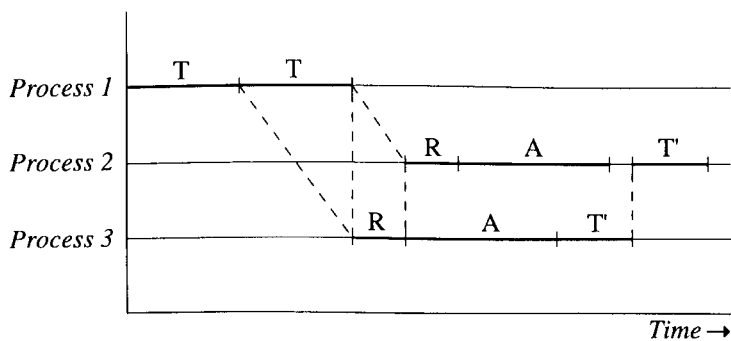


Figure 5.3b: Time-activity diagram - scenario 1

It can be seen from this diagram that, since T' is known to take longer than R , one of the processes is blocked for $Tt'-Tr$ time. The diagram illustrates that the minimum possible execution time in this scenario is $2Tt+2Tt'+2Tr+Ta$. Of course, this does not take into account the extra overheads mentioned at the start of this section, so in practice the time taken is slightly longer.

These overheads can be measured experimentally. Figure 5.3c illustrates a fragment of a MeDaL program representing scenario 1. The actor C transmits two items of data to depth actor A . Two copies of A then fire each spending time Ta processing, then transmitting a further item.

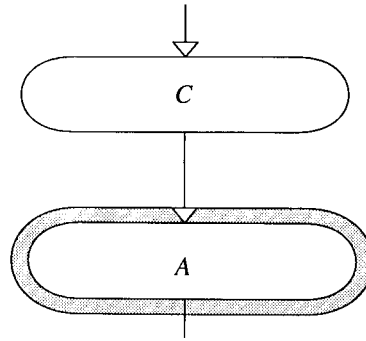


Figure 5.3c: MeDaL diagram - scenario 1

A parallel program representing this MeDaL program was executed using the MeDaL run-time system, and the elapsed time was measured from the point when actor C first called `Transmit`, to the completion of the second `Transmit` called by an instantiation of actor A , thus measuring $2Tt+2Tt'+2Tr+Ta$, plus any extra overheads incurred.

In the experiments with this program, the time Ta was varied from 0 to 5000 microseconds, in steps of 500, to find out whether any extra overhead occurred depending on the time taken by actor A . The results are shown below.

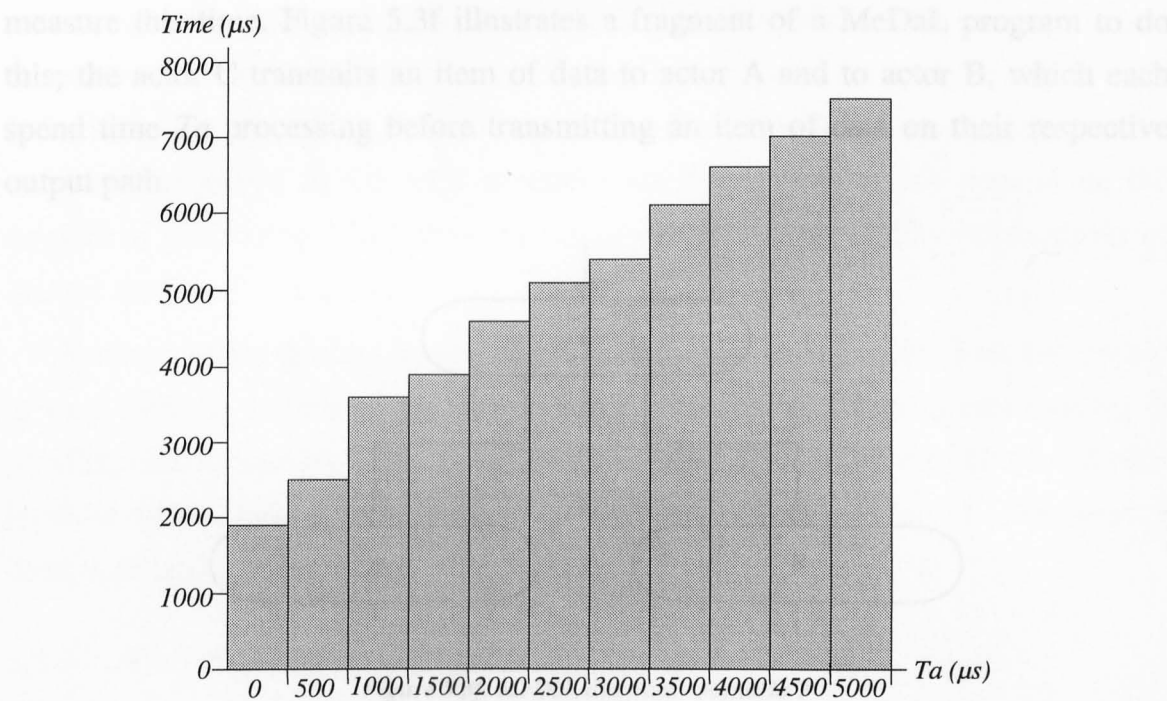


Figure 5.3d: Timings - scenario 1

As the results show, from 1000μs upwards, the total execution time recorded increases in steps of roughly 500μs, suggesting that the run-time system overheads involved in this scenario are constant.

Turning to scenario 2, Figure 5.3e illustrates the activity of the three processes in this case. Since the two Transmit functions operate on different datapaths, the corresponding Receive functions can start immediately.

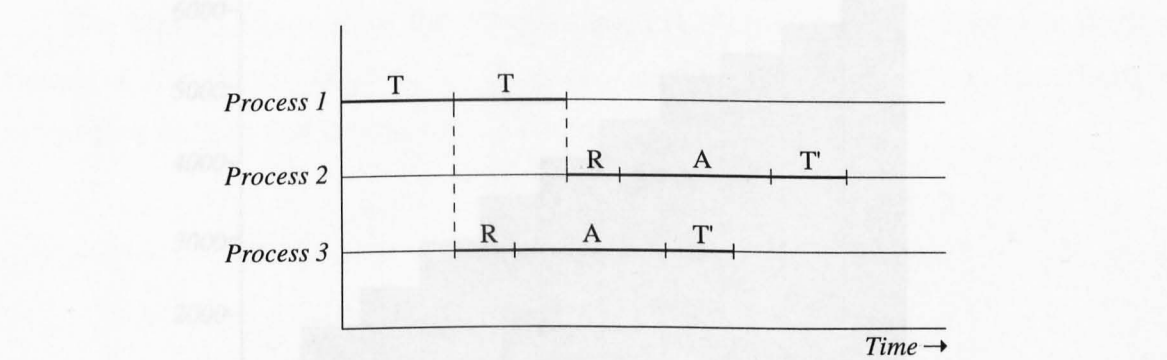


Figure 5.3e: Time-activity diagram - scenario 2

It can be seen from this diagram that the total minimum possible execution time in this scenario is $2Tt+Tt'+Tr+Ta$. Again, an example program was constructed to

measure this time. Figure 5.3f illustrates a fragment of a MeDaL program to do this; the actor C transmits an item of data to actor A and to actor B, which each spend time Ta processing before transmitting an item of data on their respective output path.

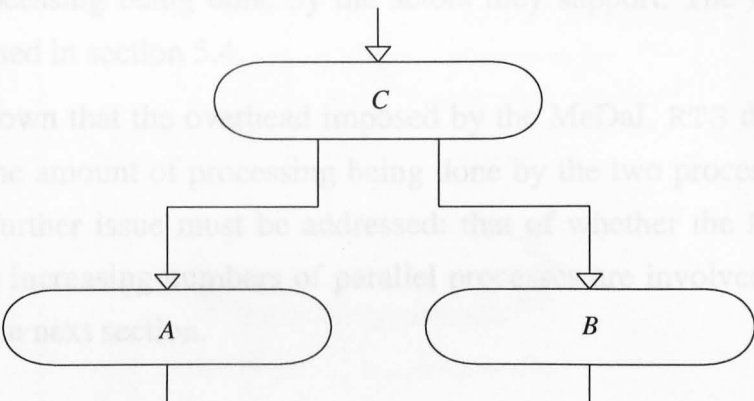


Figure 5.3f: MeDaL diagram - scenario 2

The time taken between the start of the first of the two Transmit calls made in C, until both A and B had completed their Transmit call, was measured, thus measuring $2Tt+Tt'+Tr+Ta$ as described above, plus the extra overheads incurred. Again, the time Ta was varied from 0 to 5000 microseconds in steps of 500. Table 5.3g shows the results obtained.

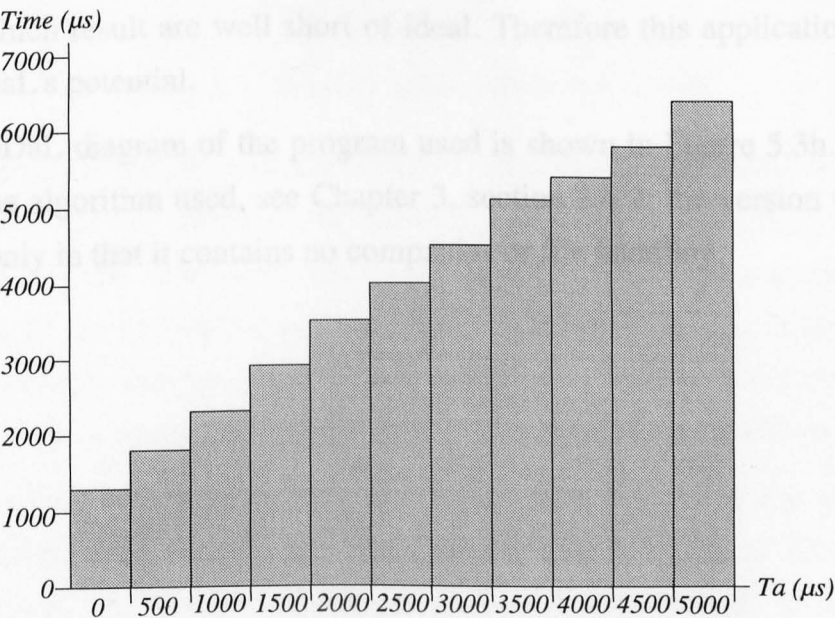


Table 5.3g: Timings - scenario 2

As one would expect, the times recorded are somewhat smaller for scenario 2 than those for scenario 1. Again, however, the timings obtained for each Ta can be seen to increase in steps of roughly 500 μ s, adding further evidence to suggest that the overheads involved in the RTS functions are fixed, and do not depend on the amount of processing being done by the actors they support. The implications of this are discussed in section 5.4.

Having shown that the overhead imposed by the MeDaL RTS does not appear to vary with the amount of processing being done by the two processes running in parallel, one further issue must be addressed: that of whether the RTS overheads increase when increasing numbers of parallel processes are involved. This issue is dealt with in the next section.

5.3.3 Varying Numbers of Actors

In order to examine the behaviour of the MeDaL RTS in a situation in which both the time taken to execute an actor and the number of actors varies, timing measurements were taken from one of the example programs coded using MeDaL. A simplified form of the matrix multiplication example given in Chapter 3 was chosen, because it is a computational problem which is easily scaled to whatever size is desired. This ease of scalability makes it a tough test for any parallel programming system, since if the parallel system does not scale so well, the speedups which result are well short of ideal. Therefore this application is a good test of MeDaL's potential.

The MeDaL diagram of the program used is shown in Figure 5.3h. For further details of the algorithm used, see Chapter 3, section 3.4.2; the version used here is simplified only in that it contains no companies or file handling.

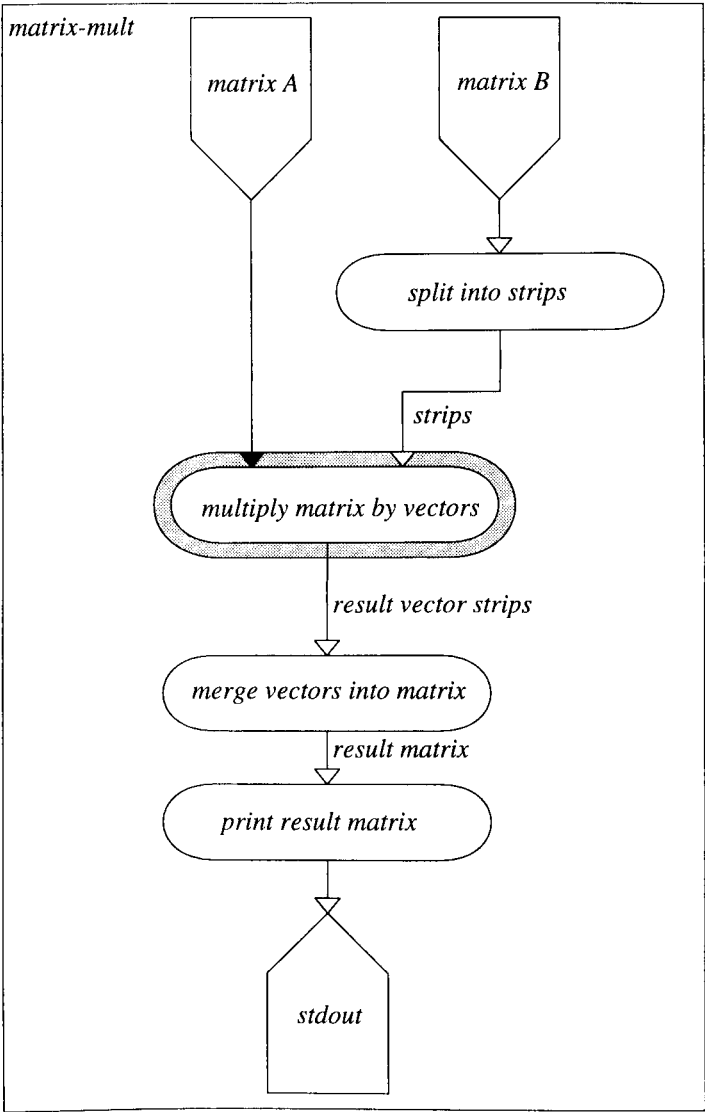


Figure 5.3h: Simplified matrix multiplication example

As in the program scenarios in the previous section, measurements were made of the time taken for the whole of the parallel portion of the program; specifically, from the start of the first Transmit operation on the datapath marked *strips* in Figure 5.3h, to the end of the last Transmit operation on the datapath marked *result vector strips*. Further, the RTS was modified so that the actor marked *merge vectors into matrix* was not allowed to fire, to avoid interference effects.

In the timing experiments with this program, the sizes of the two matrices being multiplied were varied from 625 elements (25x25) each to 22500 elements (150x150) each. The number of processes allowed to participate in the computation, n , was varied from 1 to 8. For each experiment, the second matrix B was split up into n "strips" of columns to be multiplied by matrix A; thus for each value of n ,

the depth actor fires n times, receiving a strip of vectors to be multiplied by matrix A.

Naturally, in the case of $n=1$, the code involved executes completely sequentially. However, since matrix B is passed on in only one strip, the minimum of RTS basic functions are called; `Transmit` is called only once on each datapaths. When $n=2$, `Transmit` is called twice on each path, and so on.

Thus, by comparing the times taken for varying numbers of matrix multiplication actors to the time taken by one actor, it is possible to derive an understanding of any loss of efficiency caused by the use of basic RTS functions. This comparison is a measurement of "speedup" as discussed at the start of this chapter - the "serial" version in this case being the same MeDaL program running on a single processor.

However, the speedup figure for a given array size and a given number of actors contains more information than just the unparallelisable, serial fraction of the algorithm used. The amount by which a speedup figure is less than the ideal, linear speedup figure, provides a good indication to the efficiency of the parallel program; the patterns in which these speedups vary can illustrate whether efficiency increases or decreases as the number of actors and array size vary.

An easy way to identify these patterns is to examine a visual representation of the data. Figure 5.3i shows a three-dimensional bar chart of the data obtained by experimentation, plotting array size against number of processes against speedup.

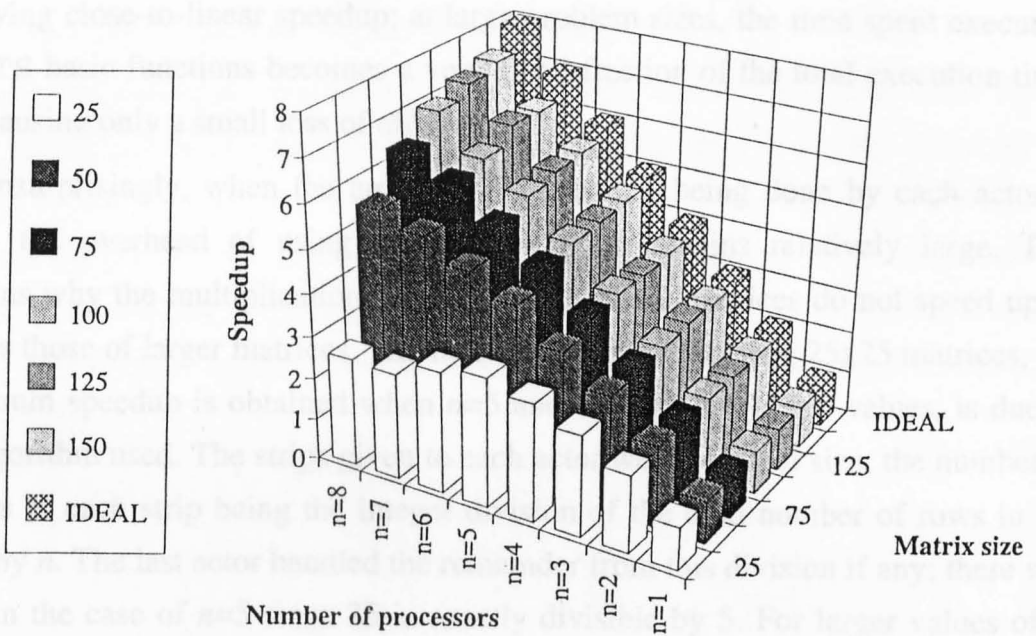


Figure 5.3i: Matrix multiplication - speedup against sequential MeDaL program

For comparison, the "ideal" of linear speedup is also shown on the graph. Thus, where $n=4$, the ideal speedup would be 4; the graph shows that with an array size of 25x25, the speedup was some way short of this; with 50x50 arrays, the speedup was closer to 4; and for all larger array sizes, the speedup was very close to 4.

In fact, a clear general trend illustrated by the graph is that the larger the array size, the closer to linear the speedup becomes. In other words, the conclusion of the previous section - that no further overheads are incurred by increasing the amount of processing being done - holds.

The other trend illustrated by the graph is that, for each array size over 25x25, adding each extra actor, i.e. each step from n to $n+1$, results in a similar speedup for each n and $n+1$. These steps are slightly larger for larger array sizes, but each matrix size over 25x25 shows a smooth, almost linear progression from small to large values of n .

What this pattern suggests is that, above a certain data (and hence processing) size, extra MeDaL actors may be employed without incurring any unforeseen overheads, in other words without any significant loss of efficiency. Use of the

MeDaL RTS for parallelism within a program does not place any barrier to achieving close-to-linear speedup; at large problem sizes, the time spent executing the RTS basic functions becomes a very small fraction of the total execution time, thus causing only a small loss of efficiency.

Unsurprisingly, when the amount of processing being done by each actor is small, the overhead of using the MeDaL RTS remains relatively large. This explains why the multiplications of 25x25 and 50x50 matrices do not speed up as well as those of larger matrices. The fact that when multiplying 25x25 matrices, the maximum speedup is obtained when $n=5$ and declines for larger values, is due to the algorithm used. The strips given to each actor were of equal size; the number of vectors in each strip being the integer division of the total number of rows in the array by n . The last actor handled the remainder from this division if any; there was none in the case of $n=5$ since 25 is exactly divisible by 5. For larger values of n , each actor was doing small amounts of work, with the last actor doing even less. Thus, the overhead imposed by the RTS became more significant, and a drop in speedup was observed, in just the same way as in the scenarios described in section 5.3.2.

The alternative way of measuring speedup, mentioned at the start of this chapter, is to measure gains in execution time not against a version of the same (MeDaL) program running on one processor, but against a purely serial program, using essentially the same algorithm but without any parallel constructs. As stated earlier, the basic algorithm involved in matrix multiplication scales very well. When processing larger matrices, the only overhead incurred (above the extra multiplication instructions) is to execute iteration constructs. However, any parallel version of this algorithm must partition the matrices, and transmit these partitions to the worker processes; and the larger the matrices, the greater this partitioning and transmitting task becomes. Moreover, because this partitioning must be done before the partitions can be processed, it is not parallelisable; so if it is included in the measurement of total time taken, it represents an increasingly large overhead. Thus, as matrix size increases, speedup would not increase as fast as it would if this overhead was fixed. However, as the overhead of partitioning only increases proportionally to the number of elements, while the time taken for the multiplication increases proportionally to the cube of this number, partitioning represents

an increasingly small serial fraction of the total computation, and so one would expect speedups to increase even if only gradually.

Speedup results for matrix multiplication were derived using the same figures used in the previous experiment, but compared against a purely serial matrix multiplication program processing matrices of the same range of sizes. Because the time for the MeDaL version was measured from the first `Transmit` operation on the datapath labelled *strips* in Figure 5.3h, the time for partitioning is included. The graph of speedups against number of processors against matrix size is shown in Figure 5.3j.

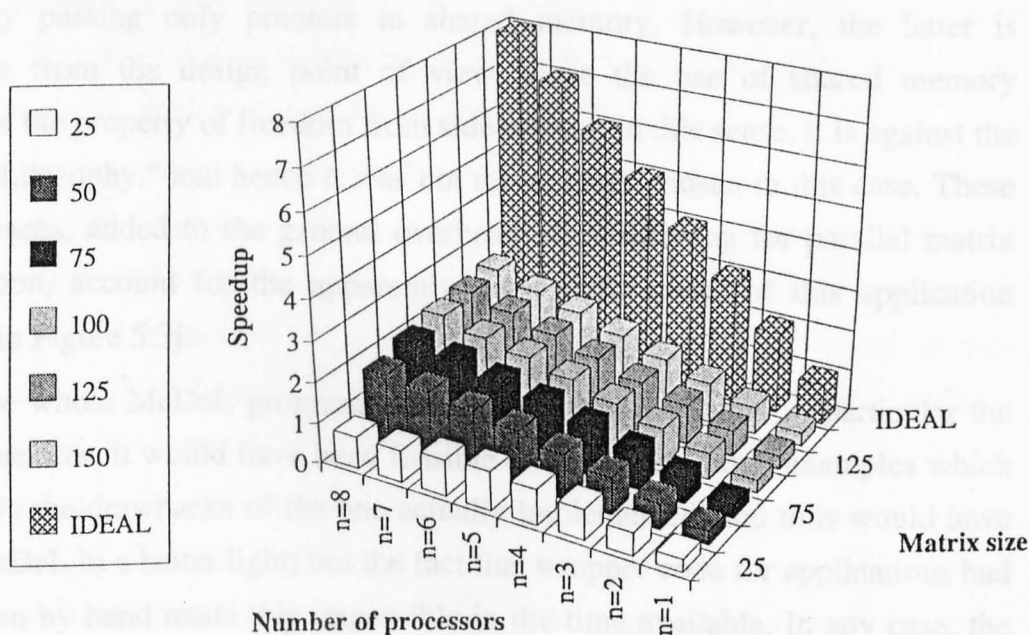


Figure 5.3j: Matrix multiplication - speedup against serial program

This speedup graph shows the expected behaviour; that as matrix size grows, so too does the speedup increase; but more slowly than in the previous example as the overhead of partitioning also increases. When more processors are used, the partitioning process again grows, and so the graph is flatter on both these axes. The fact that speedups do increase with larger matrices and more processors illustrates that the overhead of using the MeDaL RTS does not become prohibitive across a range of grain sizes (and experiments with larger arrays and more processors

suggested that this pattern continues). Nevertheless, comparison with linear speedup shows that this is a fairly poor result.

However, this result is a reflection more on the implementation of matrix multiplication used, as much as on the RTS implementation. Apart from the fact that serial matrix multiplication does not suffer from an increasing partitioning overhead, the MeDaL version suffered on two counts. Firstly, the actual multiplication algorithm was written for simplicity rather than efficiency; various shortcuts were ignored in favour of a simple working implementation. Secondly, and more importantly, the most obvious strategy for partitioning was used - sections of matrix B were copied into individual arrays representing the strips to be processed by the deep actor. This copying is relatively time-consuming, and could have been avoided by passing only pointers to shared memory. However, the latter is undesirable from the design point of view, since the use of shared memory undermines the property of freedom from side-effects; in this sense, it is against the MeDaL "philosophy," and hence it was not the strategy chosen in this case. These two drawbacks, added to the general overhead of partitioning for parallel matrix multiplication, account for the apparently poor performance of this application illustrated in Figure 5.3j.

Had the whole MeDaL programming system been available, in particular the harness generator, it would have been feasible to implement other examples which did not have the drawbacks of the one actually implemented, and thus would have showed MeDaL in a better light; but the fact that wrapper code for applications had to be written by hand made this impossible in the time available. In any case, the aim of the exercise was to evaluate the efficiency of the run-time system, rather than the use of the MeDaL notation itself.

Having studied the behaviour of the MeDaL RTS implementation and a sample application, using the data presented in this section, these results can now be analysed.

5.4 Evaluation of Results

To understand the significance of the results described in the last section, it is necessary to return to the concept of grain size described in earlier chapters. Grain

size refers to the amount of processing done by a process between synchronisations - in the case of MeDaL, the amount of processing done by an actor between receiving its input data and transmitting its output data. Therefore, in considering the question posed at the start of this chapter, about whether MeDaL can be implemented efficiently enough to provide speedups for parallel programs, an alternative question can be put: is there a grain size for actors at which their execution under MeDaL is acceptably efficient?

5.4.1 Grain Size

The results shown in section 5.3.2 show that MeDaL overheads remain fixed as the amount of processing (grain size) in actors increases; thus, as stated above, the proportion of computation taken up by RTS functions decreases as the grain size gets larger. This means that, if MeDaL is efficient at any grain size, there is no upper bound on the grain size at which it is efficient.

The results presented in section 5.3.3 further show that above a certain grain size, one can add further processors to work on the computation without a significant loss of efficiency (at least, on the fairly small numbers of processors available on the Multimax). This confirms that there is no upper bound on this machine. However, the figures obtained in the experiments in both these sections clearly show that there is a lower bound.

A first guess at this lower bound is suggested by scenario 2 in section 5.3.2 - the "best case" for the RTS overhead of adding another actor. Given that the total amount of processing being done takes $2Ta$ time, the program first showed a speedup (over a sequential version) when $Ta=1500\mu s$. Thus, one could say that the minimum efficient grain size for MeDaL is $1500\mu s$ on the Multimax; to put this in a more architecture-independent context, using the figures mentioned in section 5.3.1, this is equivalent to around 540 floating-point multiplications (or 980 integer multiplications).

Using the worst case rather than the best case to determine the lower bound, the scenario 1 program began to exhibit a speedup when approximately $Ta=2500\mu s$. If this is used as a measurement of the minimum efficient grain size, then the grain size is $2500\mu s$ on the Multimax, or roughly 920 floating-point multiplications.

Of course, floating-point operations are in themselves something of a worst

case in terms of computer performance, so in a practical program, perhaps with a mix of integer and floating-point operations, the grain size (in terms of instructions) necessary for this implementation of MeDaL to yield speedups would in fact be somewhat higher. However, even if we conclude that the minimum efficient grain size of this implementation is in the range 1000-2000 instructions, this still falls clearly within what is suggested for the term "medium-grained" in Chapter 2, section 2.1.7.

5.4.2 Limitations

Identifying the minimum efficient grain size has its limitations. It is not generally possible to say that because a program has potentially parallel computations of above a minimum size it is "worth" parallelising, because of two factors:

- the desire for "acceptable" speedups; and
- the presence of unparallelisable sections of the algorithm.

The first of these two factors is hard to define qualitatively, not only because the standard of what may be "acceptable" will vary from one application to another, but also because it tends to be a fairly subjective judgement.

However, a general feel for the performance of the MeDaL RTS can be obtained by tentatively imposing some criterion of what is an acceptable speedup. For instance, if one arbitrarily suggests that only speedups which come within 25% of linear speedup are acceptable, then clearly the speedup obtained in the case of 3000 μ s grain size in scenario 1 above (see section 5.3.2, in particular Figure 5.3d) is no longer acceptable. Although MeDaL can run 6000 μ s worth of processing in only 5351 μ s, this is only a speedup of 1.121 compared to a theoretical linear speedup of 2. In this example the grain size has to rise to 5000 μ s before the speedup reaches 1.502, crossing the arbitrary 25% line at 1.5.

The second factor, that of un-parallelisable components of an algorithm, is demonstrated very clearly in the matrix multiplication example. Because timing was to include the RTS overhead of transmitting strips of vectors to the depth actor, the start time used was the start of the first call to `Transmit` made by the actor which splits matrix B into strips. However, this meant that the total time recorded included the time taken by that actor to generate the other strips (i.e., to partition that data). As the algorithm used was chosen for simplicity rather than efficiency,

for the smaller matrices the overhead imposed by this serial algorithm added a significant time overhead to the total execution time, contributing to the poor speedups seen for the smaller matrix sizes.

This point is not limited to the matrix multiplication example, but is a general feature of parallel programming: where a non-trivial algorithm is needed to partition data so that it can be worked on in parallel, this algorithm must usually be executed serially, and as such causes a further overhead on parallelism, limiting the speedups which can be achieved. Moreover, the fact that the algorithm in the parallel section must deal with partitioned data may make it more complex; this is true in the matrix multiplication example, in which the actor which does the multiplying must deal with strips and know which column of matrix B each vector is. Of course, in a complex program, it is possible that other tasks can continue while partitioning etc. is going on, and so efficiency is not significantly decreased. But in simple programs, it can be a major factor.

5.4.3 Conclusions

One general conclusion, therefore, is that in deciding whether or not it is worth parallelising a computational problem, one should take into careful account not only the grain size of the computations, but the potential for parallelism. The tighter the criteria required to make speedups acceptable, the closer attention must be paid to the algorithm used. This point is not limited to MeDaL, but can be applied to high-level programming languages in general.

Further conclusions can be drawn from the performance of the matrix multiplication example. As explained above, the lack of efficiency of the partitioning algorithm led to less than ideal scaleability in terms of the amount of processing being done. Therefore, when compared to the the serial version, MeDaL exhibited fairly poor speedups. The partitioning part of the program was inefficient for two reasons: firstly, because of the way it was coded (including copying sections of data to transmit as separate data items); and secondly, because it was inherently serial. From the first of these two points, a conclusion can be drawn in the form of a recommendation to MeDaL programmers: for maximum efficiency, when shared memory is available it should be exploited. Had the matrix-multiplication example merely passed pointers to matrix B's position in shared memory (and the multiplication actor been coded with knowledge of matrix B's storage

structure), the program would have been considerably more efficient. In other words, a tradeoff can sometimes be made between efficiency and the much-vaunted "freedom from side-effects" property of the dataflow paradigm. Either maximum efficiency, or minimum danger of side-effects can be chosen, but not both. On a shared-memory architecture, the MeDaL system does not enforce a choice one way or the other; it is for the programmer to decide, based on the particular case.

From the second point - that the partitioning section of the program was inherently serial - one final conclusion can be drawn. It was noted earlier that in a more complex program, there may be other tasks which can execute concurrently with partitioning tasks, thus masking the inefficiency of such serial partitioning algorithms. The conclusion which can be drawn from this is that MeDaL is likely to be more useful for the implementation of complex parallel programs, with many potentially concurrent sections, than for simple programs. This conclusion was expected from the outset, and once again it is a general point about the nature of high-level programming: the higher-level the language, the more inefficiencies are introduced; so the smaller the program, the less benefit can be made from using a high-level language. Because MeDaL provides horizontal, vertical and depth parallelism, and because it allows tasks to be executed as soon as resources allow, MeDaL is particularly suited to large, complex parallel programs. For such programs, whenever the number of enabled actors exceeds the number of processors, MeDaL can keep all processors busy and thus ensure maximum efficiency - this is a major benefit considering the application programmer does not have to write any parallel code to achieve this. The more complex the program, the more benefit MeDaL is likely to bring - this is as true of MeDaL as of any other programming system.

Of course, this statement is not conclusively proven. However, the experimentation with small, simple programs described in this chapter clearly indicate that the design of MeDaL should be no hindrance to the efficient execution of parallel programs. It has been shown, in particular, that:

- executing two actors is efficient so long as the actors are each doing a certain minimum amount of processing, consistent with the "medium-grain" size;
- this minimum remains constant when the total number of actors is

increased, and when the amount of work being done by each actor is increased.

Therefore, use of the MeDaL run-time system as a means of providing parallel constructs for a program allows efficient execution in the medium-grained context - assuming the algorithm to be parallelised can itself be implemented efficiently. It should be emphasised that all of the test programs were run using a realistic (if simple) implementation of the RTS. Furthermore, the test programs were also coded in a realistic way, i.e. the actor wrapper code, although written by hand, was written in such a way as to simulate the code which would have been generated from a MeDaL graph. The fact that the module of the MeDaL system to generate this code was not implemented prohibited the testing of the RTS with more complex application; but the simple tests described here indicate that the components which would be used for more complex programs allow efficient execution within certain constraints described above. The general conclusion, therefore, is that on the strength of the implementation used, the MeDaL programming system appears to support the efficient execution of parallel programs.

Conclusions

In order to present a concise view of the conclusions which may be drawn from this work, this chapter first gives an overview of the various aspects of the research described in the previous chapters; then draws these aspects together. The possibilities for future related research which this work opens up are then discussed.

6.1 Overview

The thesis opened by describing the current state of parallel computing, and in particular how the demand for high processing performance at low cost has led to a variety of architectures which allow concurrent processing. Of these architectures, two types of MIMD machines - shared-memory and distributed-memory multiprocessors - have become particularly prevalent. These two types of multiprocessor machine are particularly attractive because they utilise standard processing elements, already manufactured in bulk, which therefore offer an attractive price/performance ratio. Furthermore, the control of parallelism on these machines is left to software rather than being implemented in hardware, which makes them general-purpose. This generality of application is (as is often the case) at the expense of some speed, since flexibility at a higher level - in this case system software - generally inhibits optimisation at a lower level, in this case hardware. However, this flexibility as well as the relatively low cost of multiprocessor machines has made them successful.

Because synchronisation on these multiprocessors is not as fast as on some specialised architectures, these multiprocessors do not offer fine-grained (inter-instruction) parallelism; rather, they offer medium-grained or coarse-grained parallelism. Medium-grained parallelism is typically exploited by the use of specifically-designed parallel programming languages which map onto the constructs for parallelism provided by the machine's operating system.

These parallel programming languages come in a variety of models. Often, the model of parallelism used for a language is based upon a target architecture; for instance, many programming models assume a message-passing architecture, and base their constructs for synchronisation on the explicit transfer of data through structured channels, which map onto the inter-node links of a message-passing architecture. Alternatively, the model may assume the presence of shared resources such as semaphores which can be easily implemented on a shared-memory architecture. Indeed, many parallel programming languages are no more than well-established HLLs with constructs added to control the specific features of one architecture. The advantage of this approach is that since the parallel language code maps closely to the target architecture, the resulting code controls the machine efficiently and so achieves good processing performance. The disadvantage is that the code - and moreover the programming skills involved - are to a large extent unportable. This can only have hindered the success of parallel architectures.

Of course, a few more architecture-independent parallel languages have been developed. However, no one leading candidate has emerged, and this may be partly due to two main factors: the lack of integrated programming environments, and the fact that they are mostly text-based. To consider the first of these factors, it is necessary to remember that parallel programming is inherently more complex than sequential programming, and hence parallel programs are more difficult for the programmer to design, implement, debug and tune for performance. However, many parallel programming languages have tended to concentrate on implementation (and, to an extent, debugging). When tools for these activities are available, it is often in the form of piecemeal "toolsets" rather than integrated systems based around a common programming model. The fact that the programmer must learn and remember the user interface for each tool makes the task of programming more difficult. In the field of sequential programming, this has led to the development of integrated CASE tools which aid the programming process from design through to maintenance; however, such tools are still rare in parallel programming.

The second statement (that the textual nature of most parallel programming languages is also problematic) is also due to the complexity of parallelism. Textual languages are inherently abstract and sequential, and these factors added to the inherent complexity of parallel programming makes textual parallel languages

uniquely difficult to understand. However, textual parallel programming languages remain predominant, probably because of the relatively very recent affordability of standard graphical displays.

From these arguments it follows that there is a need for graphical parallel programming languages around which an integrated programming environment can be based. While there are a few such programming environments, not all are designed for medium-grained multiprocessor machines, and of those which are, none use the dataflow paradigm.

Yet the dataflow paradigm seems a feasible candidate for such an integrated parallel programming system. It is an inherently visual model, and has been used as the basis of sequential programming languages because it is easy to use. But because dataflow graphs contain implicit parallelism, dataflow is well suited for use as the basis of a parallel programming language. One possible solution, therefore, to the difficulty of parallel programming on medium-grained multiprocessors, might be to base a parallel programming language on the dataflow model, combining medium-grained sequential computations (written in an arbitrary sequential programming language) together using a dataflow graph.

However, existing dataflow graph notations are not well suited to the medium-grained level; many are contain specialised fine-grained operations which would be inefficient at the medium-grained level, and others contain no specialised operations at all leaving much work to the programmer.

As a result of this, it was necessary to design a new dataflow graph notation specifically for use at the medium-grained level. This notation is called MeDaL (*Medium-grained Dataflow Language*) and is based on traditional data-driven dataflow notations. As in traditional notations, data tokens flow between actors through unidirectional datapaths. The actors are subject to the strict enabling rule and contain no persistent state. As well as general-purpose actors, which contain a medium-grained computation as described above (the computation being known as the actor's method), there are four specialised types of actor which need not or cannot contain a method: the source, sink, replicator and merge actors. Because they do not contain a method, these operations are in a sense fine-grained; however, they can be implemented specially and so do not have a significant influence on efficiency.

In addition to these basic elements, MeDaL contains a number of features designed to make it an effective medium-grained programming language. These include a notation for a modular, hierarchical structure, through a system of grouping actors together into companies; a library of standard functions such as actors to handle program input/output; and provision for activity parallelism (as well as structure and result parallelism) in the form of "deep actors," multiple copies of which can fire concurrently. A further important feature of MeDaL is the ability of its datapaths to convey arbitrary structured data types, and facilities to store data items persistently relative to firings of actors. This persistent storage takes the form firstly of datapaths which retain a copy of the last data item to pass through them (F-type paths), and secondly of operations on datapaths to prevent a data item which is sent from being delivered immediately (sticky sending). The provision of persistent memory is important not only because it reduces programmer effort, but because it allows complex data types to be built up out of simpler ones, aiding working at a larger grain size.

There are various options for the way in which MeDaL programs can be compiled and executed, and these options were described in Chapter 4. It was shown how information can be extracted from the MeDaL graph and combined with the actor method code and a run-time library to form an executable program. It was illustrated that the method code can have a simple interface with the MeDaL system using encapsulation features of the C++ language.

The run-time library is central to the efficient execution of MeDaL programs, and the way in which this can be implemented on both distributed-memory and shared-memory architectures was described. The fact that this architecture-dependent detail is implemented in the MeDaL run-time library and the code generated from the MeDaL graph - rather than in the code supplied by the programmer - supports the aim of MeDaL to provide a portable parallel programming environment. The efficiency issues involved in implementing MeDaL on both types of architecture were described in detail, since efficiency is crucial to the success of any parallel programming language; programming techniques devised to address these issues were given.

Having considered the implementation of MeDaL as a programming language in theory, the final step was to evaluate the implementation in practice. Efficiency

being the focus, the performance of an implementation of the MeDaL run-time system was examined. Due to time limitations, this part of the practical work was restricted to the Multimax shared-memory architecture. Chapter 5 described the implementation and the results of experimentation with certain key parts of the run-time system; namely the functions pertaining to the transmission of data items onto datapaths, and the firing of actor methods. The time taken by these functions was measured not only in isolation, but within the context of best-case and worst-case usage scenarios of MeDaL program fragments. Within the context of these scenarios, the overheads imposed by the functions of the MeDaL run-time system were shown to be approximately constant. Further experimentation using an example application revealed that the speedups achieved by MeDaL programs increase in a steady progression when the number of participating actors, and the amount of work done by each actor, is varied. This suggests that MeDaL could be used for programming a variety of applications without incurring unexpected overheads leading to loss of efficiency. Having identified the precise effect on efficiency of using MeDaL as an implementation system, it was shown that MeDaL allows efficient execution (in the sense that it delivers speedups) when the computations involved are of 1,000 instructions upwards. Thus, MeDaL was demonstrated to be a viable candidate for use as a medium-grained parallel programming language, in the terms defined in Chapter 2.

6.2 Conclusions

The above overview of the preceding chapters of this thesis provide the basis for a summary of the conclusions which can be drawn.

The motivation for this work results from the research described in Chapter 2. The general conclusion is that programming languages for the multiprocessor architectures described mainly operate at a low level (close to architectural details), do little to alleviate the conceptual difficulties of parallel programming, and are not well supported by programming tools. However, a second conclusion is that the field of serial programming has clearly benefitted from abstraction, for reasons both of portability and ease of understanding. From this arose the interest in providing a higher-level parallel programming system designed for these multiprocessors. Visual programming seems to have much to offer from the point

of view of abstraction in programming languages, and the dataflow paradigm is an example of a very visual model from which it is straightforward to extract potential parallelism. Thus, the final conclusion of the research into parallel programming is that dataflow offers possibilities for a visual parallel programming system which have not previously been explored on medium-grained multiprocessor machines.

Therefore, the aim of the project became to design a dataflow language for medium-grained parallel programming, and in conjunction with this, to develop techniques which allow the efficient execution of programs expressed in this language. In designing the dataflow language, MeDaL, a number of points became clear, such as the need for a small number of efficient primitives, and a modular structure to prevent any one graph becoming unmanageably large and complex. However, more important than either of these are the issues of persistent memory and synchronisation. It was found that allowing complex data types to be built up from simpler ones, and making provision for actors to deal with data streams containing unequal numbers of data items, make the language much more flexible: not only can many algorithms be expressed more concisely, but the inclusion of these features in the language remove the need for the programmer to implement them when needed. Of course, the solutions to these problems incorporated into the language are only part of a range of possible solutions, and several of the alternatives given in Chapter 3 are also clearly viable solutions. In particular, the option of allowing actors to contain persistent state, and the provision of synchronous and asynchronous datapaths, seem to be realistic propositions.

The design of the MeDaL language, and the subsequent development of the techniques which make it possible to implement efficiently, was to some extent an iterative process. For each language feature, it was first decided what would be needed, and then the possible implementation considered; and the notation was often then modified in the light of what seemed practical. An example of this is the deep actor, which is a compromise between a desired feature (provision for activity parallelism) with efficiency considerations (the need to avoid the creation of large numbers of datapaths, hence the decision that deep actors should share their input path/s) resulting in a tidy, concise notation.

Of course, the techniques described in this thesis are based on the provisions for parallel programming provided by the operating systems of the architectures

which were available. It must be recognised that other architectures may not provide the same facilities. In particular, where the hypercube's operating system provides functions which transport program code or data from any node to any other, many distributed-memory multiprocessors (such as transputer-based machines) only provide support for communication between one node and the next, through a specified link. In such cases, further techniques would need to be developed to implement actor-to-actor communication through a point-to-point network. This should not pose a major problem, since techniques for configuration and communication on distributed architectures have already been developed.

However, it must ultimately be borne in mind that the results arrived at with the shared-memory architecture do not allow hard conclusions to be drawn about the suitability of MeDaL for distributed-memory architectures. While some techniques were developed by which the MeDaL run-time system could be implemented on the hypercube, without the implementation itself, the conclusion can be no more than informed speculation. However, since dataflow maps closely to the message-passing model (with actors as nodes and datapaths as inter-node links) there seem to be strong grounds for hope that MeDaL would prove reasonably efficient.

Moreover, since one of the main aims of implementing the basic functions of the MeDaL run-time system was to deduce the actor grain size at which MeDaL can execute efficiently, it must be noted that the conclusions about grain size drawn in Chapter 5 are also specific to the Multimax. Nevertheless, since synchronisation mechanisms do not vary greatly between shared-memory architectures, it seems likely that similar results could be achieved on similar machines. Furthermore, it should certainly be possible to improve on the figures obtained with the prototype implementation, which was coded for simplicity rather than speed. For instance, little attempt was made to ensure that cache memory was used to the fullest extent possible, a factor which can have a significant effect on execution time. If the code was profiled and re-written optimising for speed, the overhead of the MeDaL run-time system could be reduced somewhat, resulting in an improvement in the actor grain size supported (namely, to support a somewhat smaller minimum grain size).

A further limitation is that a few of the features of MeDaL were not implemented in the prototype run-time system. Standard input and file handling were not implemented. These were not considered of great important because programs

considered worth implementing in a parallel language are generally cpu-bound rather than I/O-bound, and so the provision of these features would not have significantly affected grain size or overall efficiency. In addition, the company system was not fully implemented, and no test was made of recursion. The possibility of recursion was included at the design stage because it makes certain algorithms much easier to express concisely - there can be little doubt of this. However, recursion involves the creation of a new copy of the company involved at run-time, and therefore carries a significant overhead, not only of time, but more particularly of memory space. The large-scale use of recursion might approach system-dependent limitations on memory size, and so might not be the most appropriate technique. Whether this should be reflected in the notation is another matter.

The more general question which arises out of this, though, is that of for what type of program is MeDaL *not* a suitable design and implementation medium. The point was made in Chapter 5 that algorithms with an unparallelisable partitioning section do not, in general, speed up well. More generally, the greater the serial fraction of an algorithm, the less it will (on its own) benefit from implementation in MeDaL. In a sense, the fact that MeDaL imposes its own overheads means that there is not only a minimum actor grain size at which MeDaL is viable, but a minimum program size. Of course, this minimum is less easily quantifiable than the minimum grain size, since the criteria for the success of whole applications is more complex, and will vary more from one situation to another.

Conversely, MeDaL will provide benefits as a programming language when parallelising algorithms with a small serial fraction; when implementing actors with a large grain size relative to the minimums imposed by the MeDaL run-time system; and when implementing complex programs. Indeed, the more complex the program, the more benefit can be gained from MeDaL's two main strengths: firstly, its simplicity, which makes developing and maintaining complex parallel software (with many different actors and a complex web of dependencies) simpler than a textual language. This is especially true because of the fact that MeDaL does not require the programmer to write any code for controlling the overall threads of control, nor for synchronisation. Naturally, the larger the parallel program, the more effort this takes in a traditional, textual parallel language. And secondly, the

dataflow scheduling of MeDaL programs ensures that maximum possible throughput of processing tasks, since each actor joins the run queue as early as possible (i.e. when all its input data is available). Of course, this does not guarantee optimal processor utilisation, but that is a matter for the scheduler rather than for MeDaL.

Examples of applications which might prove complex and large-grained enough to benefit greatly from implementation in MeDaL might include: text formatting, which consists of many different tasks such as filling, justifying, contents and index generation, with various interdependencies; relational database querying, where a variety of relational operations take place in parallel and may feed their results to further operations; parallel make program compilation; complex graphics processes such as ray-tracing; and any computation consisting of a number of stages through which streams of data are fed. Since MeDaL is intended to be a flexible, general-purpose programming language, the possibilities are endless.

Of course, without an implementation of the full MeDaL programming system, it was not possible to experiment with the process of implementing programs in MeDaL, and it is possible that the experience gained from such experimentation might have raised further issues relating to the design of the language. Such experimentation would be a priority in any future research based on this work. The next section describes this and other possible directions for any such future work.

6.3 Future Work

Given that the prototype described in this thesis implemented only part of the MeDaL programming system, the obvious starting point for any future work would be to implement the remainder. The main modules which need to be implemented are a syntax-directed MeDaL graph editor and graph browser, and the transformation module which generates actor and company wrappers from the MeDaL graph.

The presence of these modules would allow many more trial applications to be implemented fairly rapidly, given that MeDaL can be used as a parallel harness for sections of existing sequential code. The experience gained from developing these further applications would provide feedback on whether the semantics of the

MeDaL notation were acceptable, from the point of view of implementing acceptably efficient parallel programs.

Furthermore, the availability of a prototype MeDaL programming system would make it possible to study its use by those implementing the trial applications. It would be desirable to study the cognitive models of parallelism involved when using MeDaL as a programming system. Such a study would result in an analysis of MeDaL's use as a parallel programming language, from the point of view of providing a medium for the design and implementation of parallel programs; and would answer questions on the acceptability of the features which MeDaL provides, and the way in which it provides them (its syntax) which this thesis has not been able to address.

This inquiry into the syntax and semantics of MeDaL would provide a more empirical basis for presenting MeDaL as a viable parallel programming system, replacing the assumptions on which this thesis relies. Having shown that MeDaL programs can be efficiently executed at the medium-grained level, thus providing a solid technical framework, an inquiry into its suitability as a language seems the obvious next step.

Once the above steps were complete, and any feedback from them incorporated into the notation (or indeed into the design of the system as a whole), the next stage would be to design and implement further programming tools based on the graphical notation. The bulk of these would be needed to support the debugging and performance tuning phases of development. Among the possibilities would be tools to support the placing of breakpoints in a graph (using an appropriate visual metaphor such as a bar across a datapath), and checkpointing of MeDaL programs. Single-stepping (executing one actor at a time) and fragment execution (executing a subset of a graph, for testing purposes) should also be supported. Naturally, these functions would require support in the MeDaL system as well as being part of the graph editing/browsing system.

There are many other possibilities for graphical tools. Tools for data and process visualisation would greatly aid in the understanding of how a program behaved during execution. Data visualisation could include notations for visualising memory usage in datapaths (such as through colour - for instance progressing from blue for an empty datapath through to red for high memory usage

datapaths - or through the thickness of lines). It should also be possible to allow the programmer to select a particular data item, or group of items, in a datapath queue and examine their contents. Process visualisation could include displaying which actors were running, enabled or not enabled by the use of colour or icons; and colour or some graph form (such as bar charts or dials) to represent how many times a particular actor has fired, and/or how long was spent executing that actor (for one firing of the actor or a group of firings).

Further tools to be based around MeDaL could include utilities for the detection of nondeterministic constructs and race conditions at run-time, and tools for extracting code from existing sequential applications and harnessing such code into the form of a MeDaL program. Support for a range of languages for actor implementation should be developed to aid this. At the same time, tools should be provided for developing applications from scratch using MeDaL; indeed, there seems no inherent reason why tools for software project support, and for program documentation should not be integrated into the same graphical framework. After all, if the method code of an actor can be made available by (for instance) clicking on its graphical representation, there is no reason why the documentation for that module should not be made available the same way. More general documentation about a section of the program could be linked to the company level, and so on.

With a rich set of tools based on the MeDaL notation, supporting the whole software development process, MeDaL would become an example of the type of integrated visual software development environment which the field of parallel programming currently lacks.

6.4 Closing Remarks

In order to make the first step towards this goal of an integrated visual parallel programming environment, it was necessary to take elements from several traditionally separate areas of research. Ideas were taken from the fields of parallel hardware development; parallel programming; visual programming; and dataflow research, which has generally been regarded as a separate branch of each of these fields. The original contribution of this thesis is to explore the synthesis of ideas from these fields.

Within the context of these research fields, this thesis adds nothing to that of parallel hardware, indeed the intention was always to make the programming of existing platforms easier and more accessible. This thesis also adds little to the field of visual programming; MeDaL is just one variant in a set of existing visual languages (intended for sequential processing) based on the dataflow paradigm. However, this thesis does seek to add to the field of dataflow software research, by proposing new constructs which make dataflow efficient at the medium-grained level; and by proposing a potential solution to the difficulty of parallel programming, this thesis aims to contribute to that field as well.

It could be argued that proposed languages such as MeDaL, which seek to operate on a "higher level" than existing languages, can never be successful. It was mentioned at the start of this chapter that implementation at a higher level generally involves imposing overheads over an implementation at a lower level, thus reducing efficiency. This has certainly proven true with MeDaL, which executes programs less efficiently than if they were implemented in the level below, the level which the MeDaL run-time system itself manipulates. However, *reductio ad absurdum* shows that if this were the only criterion, all software applications would be implemented in machine code since it is the most efficient! Yet this is not the case. The fact is that abstraction away from low-level details makes possible software projects which would be infeasible if low-level implementation was used; the loss of efficiency is acceptable in order to make possible larger, more complex software. Computing science history shows a clear trend towards ever-greater complexity in software, and a parallel trend in increasingly high-level programming systems to solve the problems involved. The MeDaL system proposes one path towards a solution to problems which, unless trends change drastically, will become increasingly evident in the future.

Bibliography

- [Acke82] Ackerman, W.B.: "Data Flow Languages" *IEEE Computer*, vol. 15, no. 2, February 1982.
- [Agha86] Agha, G.: *Actors: a model of concurrent computation in distributed systems* MIT Press, Cambridge MA, 1986.
- [Ahuj86] Ahuja, S., Carriero, N. and Gelernter, D.: "Linda and Friends" *IEEE Computer*, vol. 19, no. 8, August 1986.
- [Alle85] Allen, J.R. and Kennedy, K.: "A Parallel Programming Environment" *IEEE Software*, vol. 4, no. 2, July 1985.
- [Ambl89] Ambler, A.L. and Burnett, M.M.: "Influence of Visual Technology on the Evolution of Language Environments" *IEEE Computer*, vol. 22, no. 10, October 1989.
- [Amda67] Amdahl, G.M.: "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities" *AFIPS Conference Proceedings*, vol. 30, pp. 483-487, 1967.
- [Andr83] Andrews, G.R. and Schneider, F.B.: "Concepts and Notations for Concurrent Programming" *ACM Computing Surveys*, vol. 15, pp. 3-43, March 1983.
- [Aral88] Aral, Z. and Gertner, I.: "Non-intrusive and Interactive Profiling in Parasight" *SIGPLAN Notices*, vol. 23, no. 9, ACM, September 1988.
- [Arvi77] Arvind, Gostelow, K.P. and Plouffe, W.: "Indeterminacy, Monitors and Dataflow" *Proc. 6th ACM Symposium on Operating Systems Principles*, ACM, November 1977.
- [Arvi90] Arvind and Nikhil, R.S.: "Executing a Program on the MIT Tagged-token Dataflow Architecture" *IEEE Trans. on Computers*, vol. 39, no. 3, pp. 300-318, March 1990.
- [Baba91] Babaoglu, O. *et al*: "Paralex: An Environment for Parallel Programming in Distributed Systems" *Technical Report UB-LCS-91-01*, Uni. di Bologna, February 1991.

- [Babb82] Babb, R.G.: "Data-Driven Implementation of Data Flow Diagrams" *Proc. 6th Int'l Conference on Software Engineering*, IEEE, September 1982.
- [Back78] Backus, J.: "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs" *CACM*, vol. 21, no. 8, pp. 613-641, August 1978.
- [Barn68] Barnes, G.H. *et al*: "The Illiac IV Computer" *IEEE Trans. on Computers*, vol. C-17, pp. 746-757, 1968.
- [Bers87] Bershad, B.N., Lazowska, E.D. and Levy, H.M.: "Presto: A System for Object-Oriented Parallel Programming" *Technical Report No. 87-09-01*, Univ. of Washington, September 1987.
- [Bic87] Bic, L.: "A Procedure-Oriented Model for Efficient Execution of Dataflow Programs" *Proc. 7th Int'l Conference on Distributed Computing Systems*, pp. 467-475, IEEE, April 1981.
- [Boon88] Boontree, K. and Lewis, E.: "Grain Size Determination for Parallel Processing" *IEEE Software*, vol. 5, no. 1, January 1988.
- [Bond89] Bondavalli, A. and Simoncini, L.: "Dataflow-like model for robust computations" *Journal of Computer System Science and Engineering*, vol. 4, no. 3, pp. 176-184, July 1989.
- [Brin77] Brinch-Hansen, P.: *The Architecture of Concurrent Programs*, Prentice-Hall International, 1977.
- [Broo84] Brookes, S.D. and Hoare, C.A.R.: "A Theory of Communicating Sequential Processes" *Journal of the ACM*, vol. 31, no. 3, July 1984.
- [Brow85] Brown, G.P. *et al*: "Program Visualisation: Graphical Support for Software Development" *IEEE Computer*, vol. 18, no. 8, August 1985.
- [Brow85b] Brown, M.H.: "Techniques for Algorithm Animation" *IEEE Software*, vol. 2, no. 1, January 1985.
- [Brow85c] Browne, J.C.: "Formulation and Programming of Parallel Computations" *Proc. Int'l Conference on Parallel Processing*, CS Press, Los Alamitos CA, 1985.

- [Brow89] Browne, J.C., Azam, M. and Sobek, S.M.: "CODE: A Unified Approach to Parallel Programming" *IEEE Software*, vol. 6, no. 4, July 1989.
- [Bueh87] Buehrer A. and Ekandham, B.: "Incorporating Data Flow Ideas into von Neuman Processors for Parallel Execution" *IEEE Trans. on Computers*, vol. C-36, no. 12, December 1987.
- [Cann90] Cann, D.C. and Oldehoeft, R.R.: "A Report on the Sisal Language Project" *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, December 1990.
- [Card87] Cardelli, L.: "Building User Interfaces by Direct Manipulation" *Systems Research Center Report no. 22*, Digital Equipment Corporation Systems Research Center, Palo Alto CA, October 1987.
- [Carr88] Carriero, N. and Gelernter, D.: "Applications experience with Linda" *SIGPLAN Notices*, vol. 23, no. 9, ACM Press, Sep 1988.
- [Carr89] Carriero, N. and Gelernter, D.: "How to Write Parallel Programs: A Guide to the Perplexed" *ACM Computing Surveys*, vol. 21, no. 3, pp. 323-357, September 1989.
- [Catt89] Catton, D.J. and Florentin, J.J.: *Structured Design Methodologies for Parallel Programming*, Strand Software Technologies, 1989.
- [Chan87] Chang, S.: "Visual Languages: A Tutorial and Survey" *IEEE Software*, vol. 4, no. 1, January 1987.
- [Chas89] Chase, J.S. *et al*: "The Amber System: Parallel Programming on a Network of Multiprocessors" *Technical Report No. 89-04-01*, Univ. of Washington, Seattle, April 1989.
- [Cheu90] Cheung, W.H., Black, J.P. and Manning, E.: "A Framework for Distributed Debugging" *IEEE Software*, vol. 7, no. 1, January 1990.
- [Davi82] Davis, A.L. and Keller, R.M.: "Data Flow Program Graphs" *IEEE Computer*, vol. 15, no. 2, February 1982.
- [Denn78] Denning, P.J.: "Operating Systems Principles for Data Flow Networks" *IEEE Computer*, pp. 86-96, July 1978.

- [Denn74] Dennis, J.B.: "First Version of a Data Flow Procedure Language" *Lecture Notes in Computer Science*, 19, pp. 362-376, Springer-Verlag, Berlin, 1974.
- {Denn80} Dennis, J.B.: "Data Flow Supercomputers" *IEEE Computer*, pp. 48-56, November 1980.
- [Denn88] Dennis, J.B. and Gao, G.R.: "An Efficient Pipeline Dataflow Processor Architecture" Joint Conference on Supercomputing, *ACM SIGArch*, pp. 368-373, November 1988.
- [DiNu88] DiNucci, D.C. and Babb, R.G. II: "Practical Support for Parallel Programming" *Proc. 21st Int'l Hawaii Conference on System Sciences*, January 1988.
- [DiNu89] DiNucci, D.C. and Babb, R.G. II: "Design and Implementation of Parallel Programs with LGDF2" *Proc. CompCon Spring 89*, IEEE, March 1989.
- [Dijk68] Dijkstra, E.W.: "Cooperating Sequential Processes" *Programming Languages* (ed. Genuys, F.) pp. 43-112, Academic Press, New York, 1968.
- [Dong87] Dongarra, J.J. and Sorensen, D.C.: "SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs" in *The Characteristics of Parallel Algorithms*, ed. Jamieson, L.H., Gannon, D.B. and Donylass, R.J., pp. 363-394, MIT Press, 1987.
- [Dunc90] Duncan, R.: "A Survey of Parallel Computer Architectures" *IEEE Computer*, vol. 23 no. 2, February 1990.
- [Enco89] *Multimax Technical Summary*, Encore Computer Corporation, January 1989.
- [Evri90] Evripidou, P. and Gaudiot, J.L.: "A Decoupled Data-Driven Architecture with Vectors and Macro Actors" *Proc. ConvPar 90/Vapp IV, Lecture Notes in Computer Science*, no. 457, pp. 39-50, Springer-Verlag, September 1990.
- [Flynn66] Flynn, M.J.: "Very High-Speed Computing Systems" *Proc. IEEE*, vol. 54, pp. 1901-1909, December 1966.

- [Fost90] Foster, I. and Taylor, S.: *Strand: New Concepts in Parallel Programming* Prentice-Hall Inc NJ, 1990.
- [Gajs82] Gajski, D.D., Panda, D.A., Kuck, D.J. and Kuhn, R.H.: "A Second Opinion on Dataflow Machines and Languages" *IEEE Computer*, pp. 58-70, February 1982.
- [Gaud85] Gaudiot, J.L. et al: "A Distributed VLSI Architecture for Efficient Signal and Data Processing" *IEEE Trans. on Computers*, vol. C-34, no. 12, pp. 1072-1088, December 1985.
- [Gaud89] Gaudiot, J.L. and Lee, A.: "Occamflow" *Journal of Parallel and Distributed Computing*, vol. 7, no. 1, August 1989.
- [Gehr88] Gehringer, E.F., Abullarade, J. and Gulyn, M.H.: "A Survey of Commercial Parallel Processors" *Computer Architecture News*, vol. 16, no. 4, pp. 75-107, ACM, September 1988.
- [Grim87] Grimshaw, A.S. and Liu, J.W.S.: "Mentat: an Object-Oriented Macro Data Flow System" *Proceedings OOPSLA 87*, pp. 35-47, ACM, October 1987
- [Guar89] Guarna, V.A. et al: "Faust: An Integrated Environment for Parallel Programming" *IEEE Software*, vol. 6, no. 4, July 1989.
- [Gurd80] Gurd, J. and Watson, I.: "Data Driven System for High Speed Parallel Computing" *Computer Design*, parts I and II, June/July 1980.
- [Gurd85] Gurd, J.R., Kirkham, C.C. and Watson, I.: "The Manchester Data Flow Computer" *CACM*, vol. 28, no. 1, January 1985.
- [Gurd86] Gurd, J.R. et al: "Fine-Grained Parallel Computing: the Dataflow Approach" Proc. "Future Parallel Computers, an Advanced Course" (ed. Treleaven, P. and Vaneschi, M.) in *Lecture Notes in Computer Science*, vol. 272, pp. 82-152, Springer-Verlag, June 1986.
- [Gurd87] Gurd, J.R.: "Data Flow Architectures" in *Major Advances in Parallel Processing* (ed. Jesshope, C.), pp. 51-68, Technical Press, 1987.
- [Hare88] Harel, D.: "On Visual Formalisms" *CACM* vol. 31, no. 5, pp. 514-530, May 1988.

- [Hoar74] Hoare, C.A.R.: "Monitors: An Operating System Structuring Concept", *CACM*, vol. 17, no.10, pp. 549-557, October 1974.
- [Hoar78] Hoare, C.A.R.: "Communicating Sequential Processes" *CACM*, vol. 21, no. 8, August 1978.
- [Hock81] Hockney, R.W. and Jesshope, C.R.: "Parallel Computers" Adam Hilger Ltd. 1981.
- [Hopk79] Hopkins, R.P., Rautenbach, P.W. and Treleaven, P.C.: "A Computer Supporting Data Flow, Control Flow and Updateable Memory" *Technical Report Series no. 144*, Univ. of Newcastle upon Tyne, 1979.
- [Hwan85] Hwang, K. and Briggs, F.A.: *Computer Architecture and Parallel Processing*, McGraw-Hill, 1985.
- [Isod87] Isoda, S., Shimomura, T. and Ono, Y.: "VIPS: A Visual Debugger" *IEEE Software*, vol. 4, no. 3, May 1987.
- [Jaza80] Jazayeri, M. *et al*: "CSP/80: A Language for Communicating Processes" *Proc. Fall IEEE CompCon 80*, IEEE, 1980.
- [Jone85] Jones, G.: *Programming in Occam*, Prentice-Hall, 1985.
- [Jul88] Jul, E., Levy, H., Hutchinson, N. and Black, A.: "Fine-Grained Mobility in the Emerald System" *ACM Trans. on Computer Systems*, vol. 6, no. 1, February 1988.
- [Karp87] Karp, A.: "Programming for Parallelism" *IEEE Computer*, vol. 20, no. 5, May 1987.
- [Kend92] *Kendall Square Research Technical Summary*, Kendall Square Research Corporation, 1992.
- [Kosi73] Kosinsky, P.R.: "A Data Flow Programming Language" *Report RC4264*, IBM T.J.Watson Research Center, NY, March 1973.
- [Kuck81] Kuck, D.J., *et al*: "Dependence Graphs and Compiler Optimisations" *Proc. 8th ACM Symp. on Principles of Programming Languages*, pp. 207-218, January 1981.
- [Kung82] Kung, H.T.: "Why Systolic Architectures" *IEEE Computer*, pp. 37-48, January 1982.

- [Lahj91] Lahjomri, Z and Priol, T.: "KOAN: a Shared Virtual Memory for the iPSC/2 hypercube" *INRIA Technical Report*, no. 1504, September 1991.
- [Lamp80] Lampson, B. and Redell, D.: "Experiences with Processes and Monitors in MESA" *CACM*, vol. 23, no. 2, February 1980.
- [Lars84] Larson, J.L.: "An introduction to multitasking on the Cray X-MP-2 multiprocessor" *IEEE Computer*, vol. 17, no. 7, pp. 62-69, July 1984.
- [Laue79] Lauer, H.C. and Needham, R.M.: "On the Duality of Operating System Structures" *ACM Operating Systems Review*, vol. 13, no. 2, April 1979.
- [LeBl87] LeBlanc, T.J. and Mellor-Crummey, J.M.: "Debugging Parallel Programs with Instant Replay" *IEEE Trans. on Computers*, vol. C-36, no. 4, pp. 471-482, April 1987.
- [Lee87] Lee, P.A.: "Parallel Processing on the Multimax Computer System" *Major Advances in Parallel Processing* (ed. Jesshope, C.), pp. 51-68, Technical Press, 1987.
- [Matt87] Mattern, F.: "Algorithms for Distributed Termination Detection" *Distributed Computing* 2, pp. 161-175, 1987.
- [Meye83] Meyers, B.A.: "Incense: A System for Displaying Data Structures" *Computer Graphics*, vol. 17, no. 3, ACM, July 1983.
- [Mori85] Moriconi, M. and Hare, D.F.: "Visualising Program Designs through PegaSys" *IEEE Computer*, vol. 18, no. 8, August 1985.
- [Mori86] Moriconi, M. and Hare, D.F.: "The PegaSys System Pictures as Formal Documentation of Large Programs" *ACM Trans. on Programming Languages and Systems*, vol. 8, no. 4, October 1986.
- [Mour89] Mourlin, F. and Cournarie, E.: "A Graphical Environment for OCCAM Programming" *OCCAM User Group Newsletter* No. 11, July 1989.
- [Mund86] Mundie, D.A. and Fisher, D.A.: "Parallel Processing in Ada" *IEEE Computer*, August 1986.
- [Nikh89] Nikhil, R. and Arvind: "Can Dataflow Subsume von Neuman Computing?" *Computer Architecture News*, vol. 17, no. 3, pp. 262-272, June 1989.

- [Oldh84] Oldoeft, A.E. and Jennings, S.F.: "Data Flow Resource Managers and their Synthesis from Open Path Expressions" *IEEE Trans. on Software Engineering*, vol. SE-10, no. 3, May 1984.
- [Padd93] Paddon, D.J. and Chalmers, A.G.: "The Effect of Configurations and Algorithms on Performance" *Parallel Computing on Distributed Memory Multiprocessors*, Computer and Systems Sciences vol. 103, pp. 77-97, Springer-Verlag 1993.
- [Panc90] Pancake, C.M. and Bergmark, D.: "Do Parallel Languages Respond to the Needs of Scientific Programmers" *IEEE Computer*, vol. 23, no. 12, pp. 13-23, December 1990.
- [Powe83] Powell, M.L. and Linton, M.A.: "Visual Abstraction in an Interactive Programming Environment" SIGPLAN Symposium on Programming Language Issues in Software Systems, *ACM SIGPLAN Notices*, vol. 18, no. 6, June 1983.
- [Prat85] Pratt, T.W.: "Pisces: An Environment for Parallel Scientific Computation" *IEEE Software*, vol. 2, no. 4, July 1985.
- [Raed85] Raeder, G.: "A Surevey of Current Graphical Programming Techniques" *IEEE Computer*, vol. 18, no. 8, August 1985.
- [Roge88] Rogers, G.: "Visual Programming with Objects and Relations" *Proc. 1988 Workshop on Visual Languages*, IEEE, October 1988.
- [Roma89] Roman, G.C. and Cox, K.C.: "A Declarative Approach to Visualising Concurrent Computations" *IEEE Computer*, vol. 22, no. 10, October 1989.
- [Ruig90] Ruighauer, A.T.S. and Yeo, T.T.E.: "Language Support for a Semi-Dataflow Programming Environment" *SIGPLAN Notices*, vol. 25, no. 9, pp. 39-47, September 1990.
- [Rumb77] Rumbaugh, J.: "A Dataflow Multiprocessor" *IEEE Trans. on Computers*, vol. C-26, no. 2, pp. 138-146, February 1977.
- [Russ78] Russel, R.M.: "The Cray-1 Computer System" *CACM*, pp 63-72, January 1978.
- [Schn83] Schneiderman, B.: "Direct Manipulation: A Step Beyond Programming Languages" *IEEE Computer*, vol. 16, no. 8, August 1983.

- [Schw86] Schwan, K. and Matthews, J.: "Graphical Views of Parallel Programs" *ACM SIGSOFT Software Engineering Notes*, vol. 11, no. 3, July 1986.
- [Sega85] Segall, Z. and Rudolph, L.: "Pie: a Programming and Instrumentation Environment for Parallel Programming" *IEEE Software*, vol. 2, no. 6, November 1985.
- [Sega89] Segall, Z., Lehr, T. et al: "Visualising Performance Debugging" *IEEE Computer*, vol. 22, no. 10, October 1989.
- [Shar82] Sharp, J.A.: *The Programmer's Approach to Data Flow as a Basis for Parallel Programming*, PhD Thesis, Uni. of London, February 1982.
- [Shar85] Sharp, J.A.: *Data Flow Computing*, Ellis Horwood, Sussex, 1985.
- [Skil90] Skillicorn, D.B.: "Architecture-Independent Parallel Computation" *IEEE Computer*, vol. 23, no. 12, pp. 38-50, December 1990.
- [Snyd82] Snyner, L.: "Introduction to the Configurable, Highly Parallel Computer" *IEEE Computer*, vol. 15, pp. 47-56, January 1982.
- [Snyd84] Snyder, L.: "Parallel Programming and the Poker Programming Environment" *IEEE Computer*, vol. 17, no. 7, July 1984.
- [Soch89] Socha, D., Bailey, M.L. and Notkin, D.: "Voyeur: Graphical Views of Parallel Programs" *SIGPLAN Notices*, vol. 24, no. 1, ACM, January 1989.
- [Stev82] Stevens, W.P.: "How Data Flow Can Improve Application Development Productivity" *IBM Systems Journal*, vol. 21, no. 2, 1982.
- [Stok90] Stoker, M.A.: *The Exploitation of Parallelism on Shared Memory Multiprocessors*, PhD Thesis, Univ. of Newcastle upon Tyne, September 1990.
- [Suhl90] Suhler, P., Biswas, J. Korner, K. and Browne, J.: "TDFL: A Task-level Data Flow Language" *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, pp. 103-115, 1990.
- [Szaf92] Szafron, D. et al: "The Enterprise Distributed Programming Model" *IFIP Working Conference on Programming Environments for Parallel Computing*, IFIP Press, April 1992.

- [Tane78] Tanenbaum, A.: "Implication of Structured Programming for Computer Architecture" *CACM*, vol. 21, pp. 237-246, March 1978.
- [Thac87] Thacker, C.P., Stewart, L.C. and Satterthwaite, E.H.: "Firefly: A Multiprocessor Workstation" *Report No. 23*, Digital Equipment Corporation Systems Research Center, Palo Alto CA, December 1987.
- [Teit81] Teitelbaum, T. and Reps, T.: "The Cornell Program Synthesiser: A Syntax-Directed Programming Environment" *CACM*, vol. 24, no. 9, September 1981.
- [Toko87] Tokoro, M. and Yonezawa, A. (ed.): *Object-Oriented Concurrent Programming*, MIT Press, Cambridge MA, 1987.
- [Trel77] Treleaven, P.C.: "Exploiting Problem-Parallelism in Computing Systems" *Technical Report no. 107*, Univ. of Newcastle upon Tyne, July 1977.
- [Trel82] Treleaven, P.C., Brownbridge, D.R. and Hopkins, R.P.: "Data Driven and Demand Driven Computer Architecture" *ACM Computing Surveys*, pp. 93, 144, March 1982.
- [Trel84] Treleaven, P.C. and Gouveia Lima, I.: "Future Computers: Logic, Data Flow, Control Flow?" *IEEE Computer*, vol. 17, no. 3, March 1984.
- [Vali90] Valiant, L.G.: "A Bridging Model for Parallel Computation" *CACM*, vol. 33, no. 8, pp. 103-111, August 1990.
- [vonN58] Neumann, J. von: *The Computer and the Brain*, Yale University Press, 1958.
- [Wadg85] Wadge, W.W. and Ashcroft, E.A.: *Lucid, the Dataflow Programming Language*, Academic Press, London, 1985.
- [Wats72] Watson, W.J.: "The TI-ASC - A Highly Modular and Flexible Super Computer Architecture" *Proc. AFIPS Fall Joint Computer Conference*, pp 221-228, AFIPS Press, Montvale NJ, 1972.
- [Wino79] Winograd, T.: "Beyond Programming Languages" *CACM*, vol. 22, no. 7, pp. 391-401, July 1979.

- [Woo83] Woo, N.S. and Agrawala, A.A.: "The DC1 Flow Schema With The Data/Control Driven Evaluation" *Proc. 1983 Int'l Conference on Parallel Processing*, IEEE, August 1983.

Appendix A

MeDaL Classes

This appendix presents C++ classes which define the interface between the actor programmer's code and the MeDaL run-time system. Objects belonging to these classes represent data items which have been received from an actor's input datapaths, or are to be sent to an actor's output datapaths. As described in Chapter 4, these classes contain the generalised parts of the wrapper level, leaving the actor method wrapper code to contain only the application-specific code.

For simplicity, the full range of operations implemented in the MeDaL classes (mathematical operations such as addition, multiplication etc for the MeDaL Integer, boolean-algebra operations for the MeDaL Boolean) have been omitted. These are not necessary so long as the programmer can cast the variable into a built-in type to do these operations. Also, "stickiness" is omitted from the simple types; assignment results in a non-sticky send. This seems sensible in a prototype implementation because the simple types consist of only one value, whereas stickiness is intended for use in building up complex data types from simpler ones.

Note that the class presented for transmitting arrays through datapaths is a **template class**, a feature of the most recent version of C++ at the time of writing (version 3.0). This allows one class definition for any kind of array, for instance arrays of integers, floats, characters, or user-defined types. The template is instantiated when it is used in method and wrapper code. Examples of this can be seen in Appendix B. Note also that assignment to elements of MeDaL Arrays does not automatically result in the transmission of the whole array on a datapath; explicit `send` and `sendSticky` operations are provided. These are felt to be more appropriate semantics given that many elements in an array must often be changed during the execution of one actor. Array element assignment is nevertheless overloaded, in order to provide array bounds checking.


```

// medal.h
// C++ Header file defining data structures used by actors
//
// version of 18/01/93
//
// Copyright (c) 1993 Jon Harley BSc
// See main copyright notice
//

/*****
 * Standard types
 */

enum mdir_type { in , out, user }; // in 0, out 1.

enum Boolean { true , false };

/*****
 * Things used by the Run Time System - functions used and data
 * structures they need
 * These things could be a class if the RTS was in C++, but it's
 * in C so most of these things have to be in C form.
 */

typedef unsigned char Bool;
#define BOOL_T 'T'
#define BOOL_F 'F'

#define NULL 0

typedef struct aid {
    unsigned troupe ;
    unsigned actor;
} ActorId;

typedef struct aid *ActorIdP;

extern "C" {rts_getactin(actoridp , unsigned);
    unsigned RTS_GetActOut(ActorIdP, unsigned);
    void RTS_WriteMsg(unsigned, char *);
    void RTS_Shfree(void *);
    void *RTS_Shmalloc(unsigned, unsigned);

    void RTS_TxBool(ActorIdP, unsigned, Bool);
    Bool RTS_RxBool(ActorIdP, unsigned);
    void RTS_TxInt(ActorIdP, unsigned, int);
    int RTS_RxInt(ActorIdP, unsigned);
    void RTS_TxArray(ActorIdP, unsigned, void *, unsigned, Bool);
    void RTS_RxArray(ActorIdP, unsigned,void **,unsigned *,Bool *);
    void RTS_UnTxArray(ActorIdP, unsigned, void **, unsigned *);
    Bool RTS_IsHeld(ActorIdP, unsigned);
    void RTS_ActGoing(ActorIdP);
    void RTS_ActOver(ActorIdP);

/*****
 * Classes passed in and out
 */

// MeDaL boolean

class Mbool {
    Boolean truth ;
    mdir_type direction;
    unsigned whichpath;
    ActorIdP my_id;
    void transbool();
    Boolean recbool();

```

```

public:
    Mbool() { truth = true; direction = user; whichpath = 0; }
    Mbool(Boolean tree) { truth = tree; direction = user; }
    void setup(mdir_type dir, unsigned what, ActorIdP me_ptr);
    operator Boolean() { return(truth) ; }
    Mbool operator=(Mbool & black);
    Mbool operator=(Boolean & white);
    Mbool operator!();           // |, & etc. omitted for brevity
};

void Mbool::setup(mdir_type dir, unsigned what, ActorIdP me_ptr)
{
    direction = dir;
    my_id = me_ptr;

    if (direction == in) {
        whichpath = RTS_GetActIn(me_ptr, what);
        truth = rechbool();
    }
    else {
        whichpath = RTS_GetActOut(me_ptr, what);
    };
}

Boolean Mbool::rechbool(void)
{
    Bool inval ;

    inval = RTS_RxBool(my_id, whichpath);
    if (inval == BOOL_T)
        return true;
    else
        if (inval == BOOL_F)
            return false;

    // printf("Aargh! %i !", (int)inval); or something.
    return false;
}

void Mbool::transbool(void)
{
    Bool truth_val ;

    if (truth == true)
        truth_val = BOOL_T;
    else
        truth_val = BOOL_F;

    RTS_TxBool(my_id, whichpath, truth_val);
}

Mbool Mbool::operator=(Mbool & black)
{
    truth = black.truth;
    if (direction == out)
        transbool();
    return *this;
}

Mbool Mbool::operator=(Boolean & white)
{
    truth = white;
    if (direction == out)
        transbool();
    return *this;
}

```

```

Mbool Mbool::operator!(void)
{
    Boolean tt ;

    if (truth == true)
        tt = false;
    else
        tt = true;

    return Mbool(tt);
}

// -----
// MeDaL integer

class Mint {
    int intval ;
    mdir_type direction;
    unsigned whichpath;
    ActorIdP my_id;
    void transint();
    int recint();
public:
    Mint() { intval = 0; direction = user; whichpath = 0; }
    Mint(int z) { intval = z; direction = user; }
    Mint(float f) { intval = (int)f; direction = user; }
    void setup(mdir_type dir, unsigned what, ActorIdP me_ptr);
// operator int() { intval ; } // doesn't work... g++ bug
    int integer() { return intval ; } // so we must do this :-(
    operator float() { return (float)intval ; }
    operator unsigned int() { return (unsigned int)intval; }
    Mint& operator=(Mint &);
    Mint& operator=(const int &);
    Mint& operator=(const unsigned int &);
    Mint& operator=(const float &);
};

void Mint::setup(mdir_type dir, unsigned what, ActorIdP me_ptr)
{
    direction = dir;
    my_id = me_ptr;

    if (direction == in) {
        whichpath = RTS_GetActIn(me_ptr, what);
        intval = recint();
    }
    else {
        whichpath = RTS_GetActOut(me_ptr, what);
    }
};

int Mint::recint(void)
{intval ;

    inval = RTS_RxInt(my_id, whichpath);
    return inval;
}

void Mint::transint(void)
{
    RTS_TxInt(my_id , whichpath, intval);
}

```

```

Mint & Mint::operator=(Mint & choc)
{
    intval = choc.intval;
    if (direction == out)
        transint();
    return *this;
}

Mint & Mint::operator=(const int & cake)
{
    intval = cake;
    if (direction == out)
        transint();
    return *this;
}

Mint & Mint::operator=(const unsigned int & bar)
{
    intval = (int)bar;
    if (direction == out)
        transint();
    return *this;
}

Mint & Mint::operator=(const float & chip)
{
    intval = (int)chip;
    if (direction == out)
        transint();
    return *this;
}

// -----
// MeDaL array

template <class Type>
class Mary {
    unsigned size ;           // size in terms of number of elements, not bytes.
    Type *aa;
    Boolean is_held;
    Boolean keep;
    mdir_type direction;
    unsigned whichpath;
    ActorIdP my_id;
    void init(const Type*, unsigned, Boolean);
    void chuck(void);
    unsigned checkbounds(unsigned);
    void recarray();
    void yankarray();
    void transarray();

public:
    Mary(unsigned ni=0) { init(0 , ni, true); }
    Mary(const Type *ar, unsigned ni) { init(ar , ni, true); }
    Mary(const Mary &m) { init(m.aa , m.size, true); }
    ~Mary() { chuck() ; }

    void setup(mdir_type dir, unsigned what, ActorIdP me_ptr);
    unsigned getSize() { return size ; }
    void extend(unsigned);
    Boolean isSticky() { return is_held ; }
    void send();
    void sendSticky();

```

```

    Mary& operator=(const Mary&);
    Mary& operator=(const Type*);    // only for Mary<char> really.
    Type& operator[](unsigned ix) { unsigned nix = checkbounds(ix);
                                   return aa[nix]; }
};

template <class Type> void
Mary<Type>::init(const Type *array, unsigned sz, Boolean scratch)
{
    aa = NULL;
    size = 0;

    if (scratch == true) {
        is_held = false;
        direction = user;
        keep = false;
    };

    if (sz == 0)
        return;

    extend(sz);

    if (array != 0) {
        for (unsigned ix = 0; ix < size; ix++)
            aa[ix] = array[ix];
    };
}

template <class Type> void
Mary<Type>::chuck(void)
{
    if ((aa != NULL) && (keep == false))
        RTS_Shfree(aa);
}

template <class Type> unsigned
Mary<Type>::checkbounds(unsigned index)
{
    char bounds_warn[64] ;

    if ((index >= size) || (aa == NULL)) {
        if (index >= size) {
            sprintf(bounds_warn,
                    "Warning, array out of bounds (%u/%u) in t%u a%u",
                    index, size, my_id->troupe, my_id->actor);
        }
        else {
            sprintf(bounds_warn, "Dire warning! About to deref null pointer");
        };
        RTS_WriteMsg(3, bounds_warn);
        return 0;
    };

    return index;
}

template <class Type> void
Mary<Type>::recarray(void)
{
    void *bb ;
    Bool f_type;

    bb = (void *)aa;
    RTS_RxArray(my_id, whichpath, &bb, &size, &f_type);
}

```

```

    if (f_type == BOOL_T)
        keep = true;

    aa = (Type *)bb;
}

template <class Type> void
Mary<Type>::yankarray(void)
{
    void *bb ;

    bb = (void *)aa;
    RTS_UnTxArray(my_id, whichpath, &bb, &size);

    aa = (Type *)bb;
}

template <class Type> void
Mary<Type>::transarray(void)
{
    Bool please_hold ;

    if (is_held == true)
        please_hold = BOOL_T;
    else
        please_hold = BOOL_F;

    RTS_TxArray(my_id, whichpath, (void *)aa, size, please_hold);

    aa = NULL;                // maybe a new array
    size = 0;                  // of the same size
    is_held = false;           // should be created here.
}

template <class Type> void
Mary<Type>::setup(mdir_type dir, unsigned what, ActorIdP me_ptr)
{
    direction = dir;
    my_id = me_ptr;
    is_held = false;

    if (direction == in) {
        whichpath = RTS_GetActIn(me_ptr, what);
        recarray();
    }
    else {
        if (direction == out) {
            whichpath = RTS_GetActOut(me_ptr, what);
            if (RTS_IsHeld(me_ptr, whichpath) == BOOL_T) {
                is_held = true;
                yankarray();
            }
        };
    };
}

template <class Type> void
Mary<Type>::extend(unsigned extra)
{
    Type *new_aa ;
    unsigned ix = 0, new_size, typesize;

    if (keep == true) {
        RTS_WriteMsg(3 , "Warning: user is extending F-type input path.");
    };
}

```

```

new_size = size + extra + 1; // you get 1 free :-o
typesize = sizeof(Type);

if (((void *)new_aa = RTS_Shmalloc(new_size, typesize)) == NULL)
    return;

while (ix < size)
    new_aa[ix] = aa[ix++];
while (ix < new_size)
    new_aa[ix++] = (Type)0;

if (aa != NULL)
    RTS_Shfree(aa);

aa = new_aa;
size = new_size;
}

template <class Type> void
Mary<Type>::sendSticky(void)
{
    is_held = true;
    if (direction == out)
        transarray();
}

template <class Type> void
Mary<Type>::send(void)
{
    is_held = false;
    if (direction == out)
        transarray();
}

template <class Type> Mary<Type>&
Mary<Type>::operator=(const Mary &ma)
{
    if (this == &ma)
        return *this;          // copying an array to itself would be silly

    if (aa != NULL)
        RTS_Shfree(aa);

    init(ma.aa, ma.size, false);

    return *this;
}

// The following is purely for assigning constant strings to Mary<char> things.
// It would be inadvisable to use it for any other Mary<type>.

template <class Type> Mary<Type>&
Mary<Type>::operator=(const Type *s_ptr)
{
    unsigned s_len = 0;

    if (aa != NULL)
        RTS_Shfree(aa);

    if ( (s_ptr != NULL) && (sizeof(Type) == sizeof(char)) ) {
        while (*(s_ptr + s_len) != 0)
            ++s_len;
        init(s_ptr, s_len, false);
    };

    return *this;
}

```

Appendix B

Example Application

This appendix presents the code needed to implement the simplified matrix multiplication example described in Chapter 5. Three source code files are included, their function being as follows:

generic.c

This program is part of the wrapper code which would ultimately be generated from the MeDaL graph by the wrapper generator module, though in this case it was coded by hand (though, for realism, it is coded in such a way as if it had been automatically generated). Entry into the executable derived from the MeDaL program is in fact into the run-time system; after some initial setting-up, the RTS calls the function `InitCo0()` (initialise company 0, the root company) in `generic.c`. This function name is, of course, fixed at the compile time of the RTS. This function, and those it calls, "register" the actors and datapaths with the RTS (by calling RTS functions) so that the RTS can build up an initial "roadmap" of the layout of the graph. Among the information passed to the RTS in this way is a pointer to the similar initialisation function of any sub-companies, and to the wrapper function of each actor wrapper, which are contained in the next file.

wrappers.C

This module contains the wrapper functions for each actor method. Each such function is called by the EPT system when requested to do so by the MeDaL run-time system, which passes EPT a pointer to the wrapper function. The wrappers simply set up the data structures needed by the actor method functions, before calling them. Again, these wrapper functions were coded in as realistic a way as possible, to simulate their having been generated automatically from a MeDaL graph.

actors.C

This file contains the C++ source code actually written by the application programmer. The code is that used in experiments described in Chapter 5.

file: generic.c

```

/* company0: generic.c
 * generated from company0/src/* and graph info
 * by hand
 */

#include "thread.h"
#include "rts.h"

extern void c0a0(unsigned);
extern void c0a1(unsigned);
extern void c0a2(unsigned);
extern void c0a3(unsigned);
extern void c0a4(unsigned);
extern void c0a5(unsigned);

extern void RTS_WriteMsg(unsigned, char*);

Bool InitCo0(unsigned myid)
{
    CompanyP myco ;
    Bool ok;
    CActorP actr;
    CompanyP RTS_RegisterCompany(unsigned, char *, SFuncP);
    CActorP RTS_MakeActor(CompanyP, ACT_TYPE, FuncP);
    void RTS_SetActIn(CActorP, unsigned);
    void RTS_SetActOut(CActorP, unsigned);
    void RTS_MakePath(CompanyP, PATH_TYPE, unsigned, unsigned, Bool);
    void SrcCo0(unsigned);

    if ((myco = RTS_RegisterCompany(myid, "MatMul", (SFuncP)SrcCo0))
        == (CompanyP)NULL)
        return FALSE;

/* The following sets up the actors & paths. It knows about merged paths.
 * The order of actors and output paths determines their place in the roadmap.
 */

    actr = RTS_MakeActor(myco, ACT_M_SOURCE, (FuncP)c0a0);
    RTS_SetActOut(actr, 0);
    actr = RTS_MakeActor(myco, ACT_M_SOURCE, (FuncP)c0a1);
    RTS_SetActOut(actr, 1);
    actr = RTS_MakeActor(myco, ACT_GEN, (FuncP)c0a2);
    RTS_SetActIn(actr, 1);
    RTS_SetActOut(actr, 2);
    actr = RTS_MakeActor(myco, ACT_DEPTH, (FuncP)c0a3);
    RTS_SetActIn(actr, 0);
    RTS_SetActIn(actr, 2);
    RTS_SetActOut(actr, 3);
    actr = RTS_MakeActor(myco, ACT_GEN, (FuncP)c0a4);
    RTS_SetActIn(actr, 3);
    RTS_SetActIn(actr, 6);
    RTS_SetActOut(actr, 4);
    actr = RTS_MakeActor(myco, ACT_GEN, (FuncP)c0a5);
    RTS_SetActIn(actr, 4);
    RTS_SetActOut(actr, 5);
    actr = RTS_MakeActor(myco, ACT_STDOUTS, (FuncP)NULL);
    RTS_SetActIn(actr, 5);

    RTS_MakePath(myco, PATH_F, 3, MARY_SIZE, BOOL_F);
    RTS_MakePath(myco, PATH_E, 2, MARY_SIZE, BOOL_F);
    RTS_MakePath(myco, PATH_E, 3, MARY_SIZE, BOOL_F);
    RTS_MakePath(myco, PATH_E, 4, MARY_SIZE, BOOL_F);

```

```
RTS_MakePath(myco, PATH_E, 5, MARY_SIZE, BOOL_F);
RTS_MakePath(myco, PATH_E, 6, MARY_SIZE, BOOL_F);
RTS_MakePath(myco, PATH_E, 4, BOOL_SIZE, BOOL_F);

if ((myco->num_actors < 7) || (myco->num_paths < 7))
    return FALSE;

return TRUE;
}

/*****
 * The source-actor code is conglomerated into this function.
 */

void SrcCo0(unsigned as_troupe)
{
    /* there are no non-method sources */
    c0a0(as_troupe);
    c0a1(as_troupe);
}
```

file: wrapper.C

```

// company 0 actors 0-5 c++
// The MeDaL-generated bits
// see the following file, actors.C for which actor is which.
//

#include "medal.h"

// We must give these C linkage because they will be called from C.
extern "C" {
    void c0a0(unsigned) ; // called from SrcCo0 every time a troupe is
    void c0a1(unsigned); // created from this company
    void c0a2(unsigned); //
    void c0a3(unsigned); // called as threads
    void c0a4(unsigned); //
    void c0a5(unsigned); //
}

void c0a0(unsigned mytroupe)
{
    Mary<int> op0 ;
    ActorId me;
    ActorIdP meep = &me;
    void company0_actor0(Mary<int>&);

    me.troupe = mytroupe;
    me.actor = 0;

    op0.setup(out, 0, meep);

    company0_actor0(op0);
}

void c0a1(unsigned mytroupe)
{
    Mary<int>op0 ;
    ActorId me;
    ActorIdP meep = &me;
    void company0_actor1(Mary<int>&);

    me.troupe = mytroupe;
    me.actor = 1;

    op0.setup(out, 0, meep);

    company0_actor1(op0);
}

void c0a2(unsigned mytroupe)
{
    Mary<int>ip0 ;
    Mary<int> op0;
    ActorId me;
    ActorIdP meep = &me;
    void company0_actor2(Mary<int>&, Mary<int>&);

    me.troupe = mytroupe;
    me.actor = 2;

    ip0.setup(in, 0, meep);
}

```

```

    op0.setup(out, 0, meep);

    RTS_ActGoing(meep);

    company0_actor2(ip0, op0);

    RTS_ActOver(meep);
}

void c0a3(unsigned mytroupe)
{
    Mary<int> ip0 ;
    Mary<int> ip1;
    Mary<float> op0;
    ActorId me;
    ActorIdP meep = &me;
    void company0_actor3(Mary<int>&, Mary<int>&, Mary<float>&);

    me.troupe = mytroupe;
    me.actor = 3;

    ip0.setup(in, 0, meep);
    ip1.setup(in, 1, meep);

    op0.setup(out, 0, meep);

    RTS_ActGoing(meep);

    company0_actor3(ip0, ip1, op0);

    RTS_ActOver(meep);
}

void c0a4(unsigned mytroupe)
{
    Mary<float> ip0 ;
    Mbool ip1;
    Mary<float> op0;
    ActorId me;
    ActorIdP meep = &me;
    void company0_actor4(Mary<float>&, Mbool, Mary<float>&);

    me.troupe = mytroupe;
    me.actor = 4;

    ip0.setup(in, 0, meep);
    ip1.setup(in, 1, meep);

    op0.setup(out, 0, meep);

    RTS_ActGoing(meep);

    company0_actor4(ip0, ip1, op0);

    RTS_ActOver(meep);
}

void c0a5(unsigned mytroupe)
{
    Mary<float> ip0 ;
    Mary<char> op0;
    ActorId me;
    ActorIdP meep = &me;
    void company0_actor5(Mary<float>&, Mary<char>&);

```

```
me.troupe = mytroupe;
me.actor = 5;

ip0.setup(in, 0, meep);

op0.setup(out, 0, meep);

RTS_ActGoing(meep);

company0_actor5(ip0, op0);

RTS_ActOver(meep);
}
```

actors.C

```

// this is the only part the user writes:

#define VEC_SIZE 150
#define STRIPE_WIDTH 19
#define NUM_STRIPES 8

void company0_actor0(Mary<int> & out0) // source actor for matrix A
{
    unsigned sq , ix;
    sq = VEC_SIZE * VEC_SIZE;
    out0.extend(sq);

    for (ix = 0; ix < sq; ix++)
        out0[ix] = 0;

    for (ix = 0; ix < VEC_SIZE; ix++)
        out0[(ix * VEC_SIZE) + ix] = 1;

    out0.send();
}

void company0_actor1(Mary<int> & out0) // source actor for matrix B
{
    unsigned sq , ix;
    sq = VEC_SIZE * VEC_SIZE;
    out0.extend(sq);

    for (ix = 0; ix < sq; ix++)
        out0[ix] = sq - ix;

    out0.send();
}

// actor 2 splits the input array into columns and outputs each column with
// a corresponding index to say *which* column it actually is.

void company0_actor2(Mary<int> & in0, Mary<int> & out0)
{
    unsigned strip , index, row, col, outcol;

    for (index = 0; index < NUM_STRIPES; index++) {(+ 1) * STRIPE_WIDTH);
        outcol = 0;
        for (col = (index*STRIPE_WIDTH); col < ((index+1)*STRIPE_WIDTH); col++)
        {(col >= VEC_SIZE)
            out0[outcol + VEC_SIZE] = 99999; // end of last stripe may be empty
            else {(row = 0; row < VEC_SIZE; row++) {
                out0[outcol + row] = in0[col + (VEC_SIZE * row)];
            };
            out0[outcol + VEC_SIZE] = col;
        };

        outcol += (VEC_SIZE + 1);
    };

    out0.send();
}
}

```

```

// actor3 multiplies an array in in0 by a stripe in in1.

void company0_actor3(Mary<int> & in0, Mary<int> & in1, Mary<float> & out0)
{
    unsigned i , j, which, row, el, striper, resrow;
    float sum;

    out0.extend((VEC_SIZE + 1) * STRIPE_WIDTH);

    for (striper = 0; striper < STRIPE_WIDTH; striper++) {
        resrow = striper * (VEC_SIZE + 1);
        which = in1[resrow + VEC_SIZE];
        if (which != 99999) {= which * VEC_SIZE;
            for (i = 0; i < VEC_SIZE; i++) {
                sum = 0;
                for (j = 0; j < VEC_SIZE; j++) {
                    sum += (in0[row + j] * in1[resrow + i]);
                };
                out0[resrow + i] = sum;
            };
        };
        out0[resrow + VEC_SIZE] = which; // pass on which result row this is
    };
    out0.send();
}

// actor4 receives result vectors and puts them back together as an array
// by using sendSticky.

void company0_actor4(Mary<float> & in0, Mbool in1, Mary<float> & out0)
{
    unsigned index, which, col, sq, striper;
    Boolean dummy;
    dummy = in1;

    sq = VEC_SIZE * VEC_SIZE;

    if (out0.isSticky() == false) {
        out0.extend(sq+1);
        out0[sq] = 0.0; // number of stripes processed
    }

    for (striper = 0; striper < STRIPE_WIDTH; striper++) {
        which = (int)in0[(striper * (VEC_SIZE + 1)) + VEC_SIZE];
        if (which != 99999) {
            col = which * VEC_SIZE;
            for (index = 0; index < VEC_SIZE; index++) {
                out0[index * VEC_SIZE+which] = in0[(striper * (VEC_SIZE+1)) + index];
            };
        };
    };

    out0[sq] = out0[sq] + 1.0;

    if (out0[sq] < (float)NUM_STRIPES)
        out0.sendSticky();
    else
        out0.send();
}

```

```
// This actor's input is the result array, and it outputs it as a character
// stream to a Stdout actor.
```

```
void company0_actor5(Mary<float> & in0, Mary<char> & out0)
{
    unsigned row , col;
    char cbuf[32];

    out0 = "And the result is...\n";
    out0.send();

    for (row = 0; row < VEC_SIZE; row++) {
        for (col = 0; col < VEC_SIZE; col++) {
            sprintf(cbuf, "\t%3.2f", in0[(row * VEC_SIZE) + col]);
            out0 = cbuf;
            out0.send();
        };
    };

    out0 = "\nTa-dah!\n";
    out0.send();
}
```